REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
18 May 2005

3. REPORT TYPE AND DATE COVERED

4. TITLE AND SUBTITLE
Artificial potential field controllers for robust communications in a network of swarm robots

5. FUNDING NUMBERS

6. AUTHOR(S)
Dunbar, Thomas W. (Thomas Ward), 1982-

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US Naval Academy
Annapolis, MD 21402

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Trident Scholar project report no. 335 (2005)

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
This document has been approved for public release; its distribution is UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT: An active area of research in the robotics community is “swarm control,” where many simple robots work together to execute tasks which are beyond the capability of any single robot acting alone. Yet in order for the swarm members to work together effectively they must maintain a reliable and robust wireless communication network among themselves. The goal of this project was to create a motion control law which could fulfill the dual and sometimes conflicting requirements of executing a primary mission (e.g., search and rescue), while maintaining a robust mobile wireless communication network among the swarm members. The success or failure in sending or receiving a wireless message is inherently probabilistic, but the odds of successfully relaying a message increase considerably based upon the spatial arrangement of the swarm members. This imposes a variety of constraints on each robot’s motion. Each robot sending a message should: 1. maintain a line of sight to the receiving robot (esp. in an environment like a cave or bunker with dense walls); 2. stay within close proximity of the receiving robot (the range is dictated by the power of the transmitter); and 3. increase the overall redundancy of the swarm by maintaining requirements 1 and 2 for two or more receiving robots simultaneously. To this end, several artificial potential field controllers - a popular method of robotic control - have been developed in this project and simulated to determine their success in controlling the swarm. At a higher level, the project addressed the challenge of composing a motion control law to achieve the primary mission, while maintaining as many communication constraints as possible. This project included a proof-of-concept implementation of the motion control law on real robots. In addition, this project simulated and statistically analyzed the controller to determine its effectiveness at achieving the primary mission and maintaining a robust communication network. The effectiveness of the control law was seen both in simulation and experiment. Overall the robustness of the swarm was increased 200-300% in the scenarios considered.
.
.

14. SUBJECT TERMS:
Swarm robotics, Communication Networks, Artificial potential fields, Line of Sight, Redundancy

15. NUMBER OF PAGES
127

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Abstract

An active area of research in the robotics community is "swarm control," where many simple robots work together to execute tasks which are beyond the capability of any single robot acting alone. Yet in order for the swarm members to work together effectively they must maintain a reliable and robust wireless communication network among themselves.

The goal of this project was to create a motion control law which could fulfill the dual and sometimes conflicting requirements of executing a primary mission (e.g., search and rescue), while maintaining a robust mobile wireless communication network among the swarm members.

The success or failure in sending or receiving a wireless message is inherently probabilistic, but the odds of successfully relaying a message increase considerably based upon the spatial arrangement of the swarm members. This imposes a variety of constraints on each robot's motion. Each robot sending a message should:

1. maintain a line of sight to the receiving robot (esp. in an environment like a cave or bunker with dense walls);

2. stay within close proximity of the receiving robot (the range is dictated by the power of the transmitter); and

3. increase the overall redundancy of the swarm by maintaining requirements 1 and 2 for two or more receiving robots simultaneously.

To this end, several artificial potential field controllers - a popular method of robotic control - have been developed in this project and simulated to determine their success in controlling the swarm. At a higher level, the project addressed the challenge of composing a motion control law to achieve the primary mission, while maintaining as many communication constraints as possible.

This project included a proof-of-concept implementation of the motion control law on real robots. In addition, this project simulated and statistically analyzed the controller to determine its effectiveness at achieving the primary mission and maintaining a robust communication network. The effectiveness of the control law was seen both in simulation and experiment. Overall the robustness of the swarm was increased 200-300% in the scenarios considered.

Keywords: *Swarm robotics, Communication Networks, Artificial potential fields, Line of Sight, Redundancy*

Acknowledgments

There are several people whom I need to thank, for without their help I could not have completed this project: George Burton in the Machine Shop who built the camera mount; Norm Tyson in TSD for help rebuilding the battery; Professor Bishop for his help with mobile robots and the Koala robots; Professor Piepmeier who helped with the computer vision system; Professor Brown in the Computer Science Department who gave excelent advice on communicating through Linux; Pierre Bureau and all the other members of K-team in Switzerland; Robert Disque and Donald Garner of the CADIG lab and Bill Lowe with Technical Support staff, who all helped with networking the robots; the biggest thanks goes to Michael Spinks of the CADIG lab, who provided invaluable help with Samba.

Secondly, I need to thank: the Trident Committee; Dean Miller; Professor Shade; Professor Boden; and Professor Fowler.

Finally I must thank my Advisor, Professor Esposito, whose guidance and advice has helped me to fully realize this project.

Contents

List of Figures

Chapter 1

Introduction

A swarm of robots consists of many small, cheap, unsophisticated robots working together to accomplish a task more efficiently than a single robot. There are many potential applications for swarm robotics, some military applications include, but are not limited to, mine countermeasures both in water and on land [25]. Swarms are ideal for such applications because a swarm of robots can spread out and effectively cover a large area in a short time. In addition, if one robot is destroyed by a mine, the rest of the swarm continues to function. For these reasons, swarms are also excellent for search and rescue or reconnaissance operations (e.g. working in the caves of Afghanistan) [40]. For example, a search and rescue swarm [34] could be used in a disaster area such as the debris of the World Trade Center, the aftermath of an earthquake, searching through a darkened, stricken vessel, or a burning building.

Swarms of robots have several advantages over a single robot or even a small group of complex robots:

- the individual units of a swarm are usually less complex than a single robot, meaning they are cheaply and easily mass produced;
- a swarm is easier to reconfigure for new missions;
- swarms are more fault tolerant than a single robot because if one unit fails the rest of the swarm still functions; and
- the control of the robots is decentralized so there is no "master" robot that controls the rest of the swarm, allowing well designed control and communication algorithms to scale linearly as the size of the swarm increases.

Swarms are common in biological systems. Bees communicate and work together to defend the hive and produce honey. Ants, when working collectively, can move objects several times their mass. Wolves use packs to hunt and fish form schools for protection and hydrodynamic efficiency. When robots work together in a swarm they display some of the same behaviors and accomplishments that co-operative insects and animals demonstrate.

Yet in order for the swarms to work together effectively they must have reliable and robust communication among the team members. This project focuses on the theoretical design and experimental analysis of a distributed motion control methodology for communicating swarms of robots.

1 Problem statement

The success or failure in sending or receiving a wireless message between robots is inherently probabilistic. One way to increase the odds of successfully relaying a message without increasing the power required is to carefully construct the spatial arrangement of the swarm members to facilitate communication. This imposes a variety of constraints on each robot's motion; each robot sending a message should (see Figure 1.1):

1. maintain a line of sight to a receiving robot (especially in an environment like a cave or bunker with dense walls); and

2. stay within close proximity of the receiving robot (the range is dictated by the power of the transmitter).

Furthermore, a good control methodology will exploit the redundancy of the swarm to improve robustness to robot failure. To that end a third requirement can be introduced (refer again to Figure 1.1):

3. The robots can be constrained to move so that each robot is communicating (i.e. meeting objectives 1 and 2) with at least two other units at all times.

Therefore if one unit fails, the other robots are still communicating with at least one other swarm member. This ensures that if a single unit fails, no one unit becomes isolated from all other units.

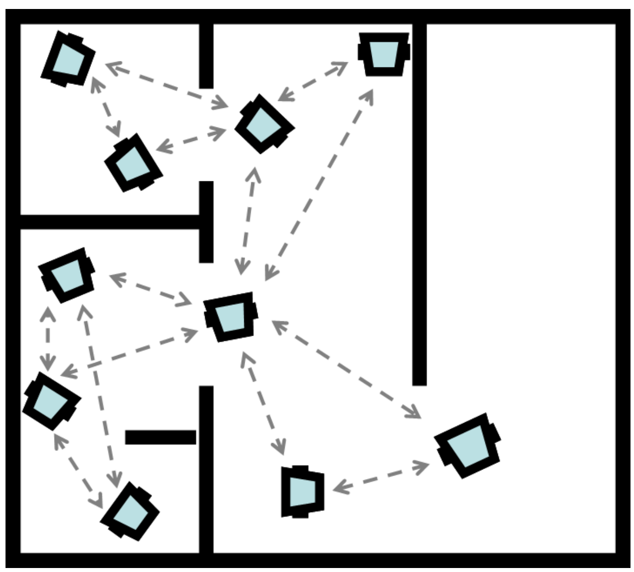


Figure 1.1: The test scenario. Maintaining multiple connections per robot, as well as line of sight and range constraints enables robust communication.

Therefore, the overall design goal of this Trident Project is to create a swarm motion control algorithm. The algorithm must meet the following criteria.

1. *Goal Completion* Compute robot velocities to allow the swarm to collectively accomplish a primary mission such as search and rescue or exploration.

2. *Robust Communication* Ensure that each individual robot maintains line of sight and proximity with at least two other robots at all times (see Constraints 1-3 above).

3. *Distributed Operation* Each robot makes its own motion control decisions based on locally available information and no single robot acts as the leader or "master" robot. A well designed distributed control algorithm would be able to scale linearly with the number of robots in the swarm.

4. *Fault Tolerant* Do not allow the failure of one robot to be detrimental to the swarm performing any of its missions

The above criteria are important because they reflect the swarm philosophy of efficiency and robustness, discussed earlier. However, this poses a significant theoretical challenge because Criteria 1:*Goal Completion* and 2:*Robust Communication* may at times be at odds with each other. Furthermore, when the algorithm is distributed according to the Criteria 3:*Distributed Operation*, there is no central "master" robot to break such deadlocks. Only a carefully designed motion control algorithm can address all of these criteria.

2 Related work to problem statement

A swarm can be defined as a group of robots moving in some collective coordinated fashion without any rigidly defined relative spatial arrangement between robots. The first work on "swarms" was provided by Reynolds [32]. This work attempted to devise a set of motion laws for a group of agents so that the collective motion would mimic that of biological system such as schools of fish and flocking birds. The resulting motion is called flocking. This work was primarily targeted at computer graphics applications. It was not considered appropriate for robotics because it did not display robust behavior in the face of unstructured or changing environments. Little work appeared in the field until Swaroop and Hedrick [35] among others ([10, 27]) began considering robot formations, primarily inspired by the automated highway project and military applications. In contrast to swarms, formation applications require that a group of robots maintain a fixed relative spatial orientation, despite movement of the group as a whole. In addition, some members of the group are arbitrarily selected as the "lead" robots, making the control scheme somewhat centralized rather than distributed, violating Criteria 3:*Distributed Operation*.

More recently there has been increased interest in swarm applications. As a result, the work of Reynolds has inspired a new round of developments in the area of flocking. Flocking implies that, while the entire group of robots does not need to maintain a fixed relative spatial orientation as in formations, the entire group is required to possess identical velocity vectors at steady state ([26, 38, 18]). The primary theoretical challenge in this work is to show collective stability (i.e. that the velocity of the members converges to some constant) and ensuring that the members do not collide. In the works cited in this paragraph it is shown that either all-to-all member information sharing is required or some centralized signal must be broadcast to guarantee stability; again this violates Criteria 3:*Distributed Operation*.

Formation and flocking approaches for motion control are not appropriate for this project because they do not afford robots the ability to operate as independent units and therefore may not be appropriate for all types of missions. Approaches requiring centralized information are not truly distributed and will not scale well (Criteria 3:*Distributed Operation*). Requiring a leader makes the swarm susceptible to single robot failure and thus this approach also fails Criteria 4:*Fault Tolerance*.

The issue of communication in mobile networks is frequently considered in wireless communication and mobile computing literature. For example, Rus [33] and Olfati-Saber and Murray [28] consider routing and consensus problems in networks with changing topologies. However, this work assumes the motion of the network nodes is not controllable and therefore they do not address Criteria 2:*Robust Communications*. Li and Rus [21] consider networking where the motions of the nodes is controllable, but only seek to create an *ad-hoc* network and not a robust communications network, again failing Criteria 2:*Robust communication*. Similarly Bishop [6] and Liu and Passino [22] consider a swarm control method that does permit deviations in individual motions but only addresses collective qualities of interest and cannot enforce Criteria 2:*Robust Communications*.

3 Project goals

The goals of this project are as follows.

Theoretical goals

- a distributed control law for maintaining an appropriate separation distance between two robots;
- a distributed control law for maintaining line of sight between two robots;
- a distributed control law for enforcing the redundancy constraint (i.e. maintaining two or more connections per robot); and
- a novel algorithm for composing the above three controllers along with the primary mission control law, in such a way that a maximum number of objectives are achieved.

Implementation and design goals

- a "proof of concept" demonstration of the operation of the swarm meeting the three design criteria outlined in Section 1, using 3 robots.

Experimental/Simulation analysis goals

- simulation of a large number of robots moving according to the control law;
- quantitative analysis of swarm performance in completing both the primary mission and connectivity and robustness in a swarm communication network.

The next section outlines specific tasks that were accomplished to successfully realize the goals and explains the overall operation of the robot and interaction of experimental components.

4 Overview of project

Figure 1.3 shows the hardware involved. Up to 8 Koala robots (Figure 1.2), each with a on-board PC, separate Motorola microprocessor, and a radio Ethernet attachment are placed in an environment populated with obstacles. In a military application, each of the robots would have a GPS receiver to give absolute position and a stereo vision setup or 360° laser range finder to determine the position of obstacles and other robots. Because that equipment is not available for this project and because the self localization problem is far beyond the scope of this work, an overhead camera interfaced to a base station PC will act as an "eye in the sky" transmitting position information to the robots. This is a standard technique in mobile robotics research and is analogous to having a satellite viewing several robotic tanks and broadcasting the tanks' position.

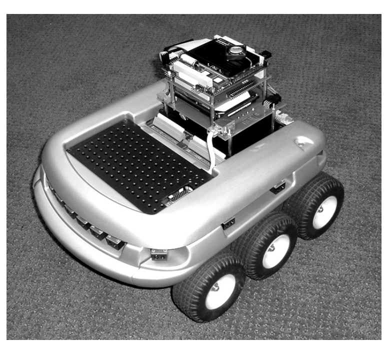


Figure 1.2: The Koala Robot.

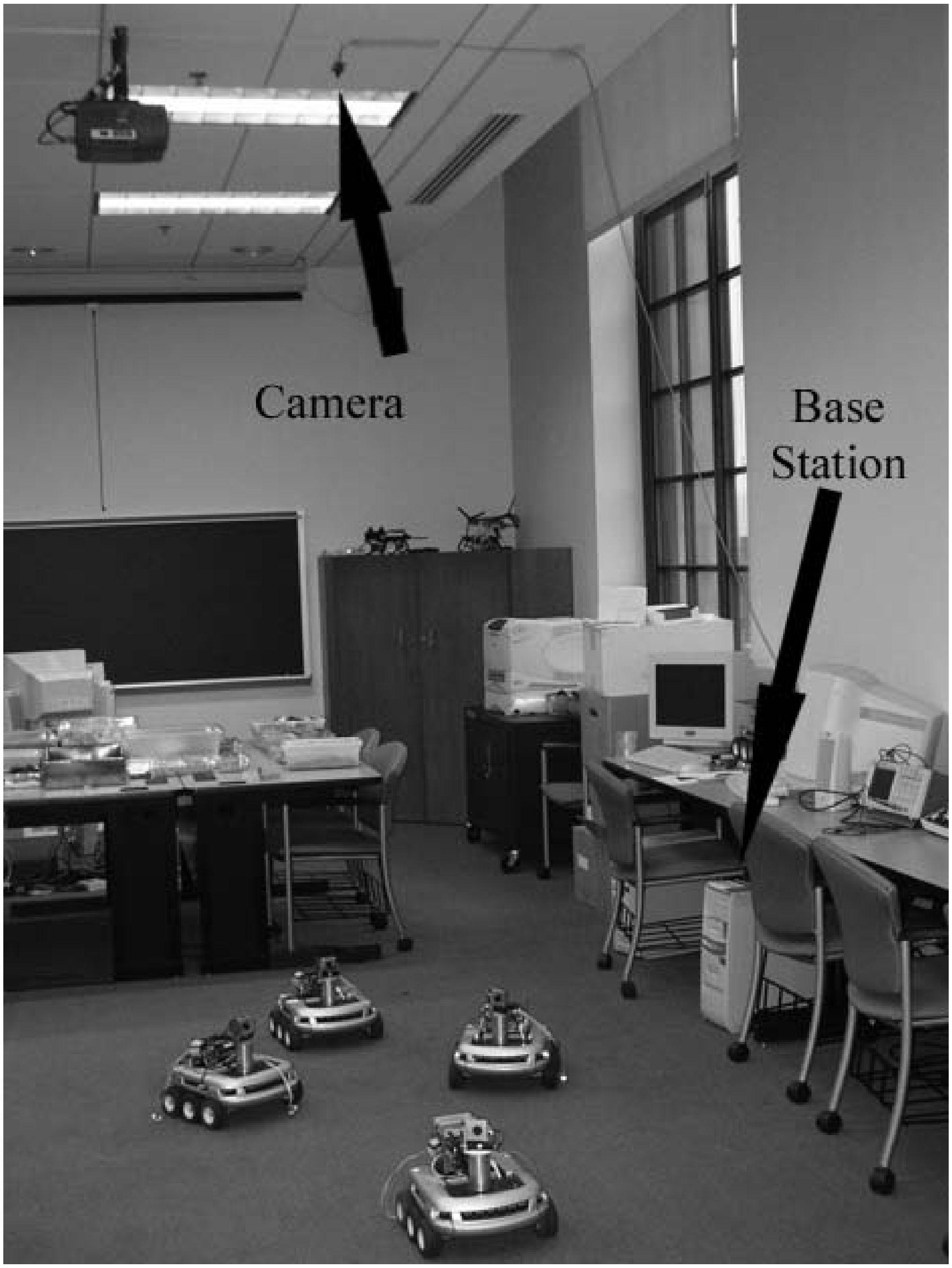


Figure 1.3: The overall experimental setup.

Figure 1.4 outlines the operation of the system. One "cycle" of the system's operation begins with an overhead digital camera taking a snapshot of the scene. The raw image must be imported to the base station PC using the MATLAB Image Acquisition Toolbox. The raw image is then processed using programs written in the MATLAB high level mathematical programming language (described in Chapter 4 Section 1.1). The positions and orientations of each robot, as well as the coordinates of the vertices of each polygonal obstacle are extracted from the image. This position information is used by five separate motion control functions: Range, Line of Sight, Redundancy, Obstacle Avoidance and Go-To-Goal. Some background on the motion control technique used in this work appears in Chapter 2 while the derivation of the three novel controllers, Range, Line of Sight and Redundancy, is detailed in Chapter 3. Each of these functions returns a suggested body velocity for the robot to achieve the respective objective. Each of these velocities is an input to the parallel composition algorithm whose function is to compose these desired motion directions so that the final desired body velocity accomplishes as many of these objectives as possible. The algorithm is described in detail in Chapter 3 Section 4. The final output velocity of the controller is then transformed into a left and right wheel velocity (see Chapter 2 Section 3). These wheel velocities are written to a file which is sent over the wireless network to the hard drive of the appropriate robot (Chapter 4 Section 2).

At each cycle, the individual robots check their hard drives for updates. If a new velocity file is found, it is read. The robot then writes these velocity values over the serial line to the embedded Motorola 68K microprocessor which sets the motor input voltages accordingly (see Chapter 4 Section 3).

The input voltages then cause the robots to move. The overhead camera takes another picture and the process repeats indefinitely.

A second phase of the project was to perform experiments at different desired numbers of communication links per robot. Simulations proved to be the most effective way to perform hundreds of tests of the controller in many situations not easily recreated in the lab. Chapter 5 Section 3 explains the experimental procedure used for this phase of the project. Additionally, the results of these experiments were quantitatively analyzed (see Chapter 5 Section 2) to determine the percentage increase in the robustness of the communication network as compared to a baseline controller (Chapter 5 Section 4).

4.1 Organization of the report

In Chapter 2, some common techniques of robot motion control are reviewed. While many of these are standard, the implementation was project-specific and a necessary step toward the end-state.

In Chapter 3 the primary theoretical contributions of the project are reported. The Range, Line of Sight and Redundancy control laws are derived and verified through computer simulation. Finally, the parallel composition scheme for composing all of the various desired velocities is discussed. Each of these functions are a novel stand alone contribution to the field.

In Chapter 4 the experimental testbed design is described in detail. Many of the experimental components were not designed to operate in tandem. Additionally, several project specific design solutions were created to achieve the necessary experimental setup.

In Chapter 5 the experimentation done via simulation is discussed. This chapter includes a detailed explanation of the various metrics used to determine and quantify the robustness of the swarm communication network. The results of the various experiments are documented and interpreted.

Chapter 6 reflects on the contributions and tasks achieved in this project. In addition, ideas for future research are examined.

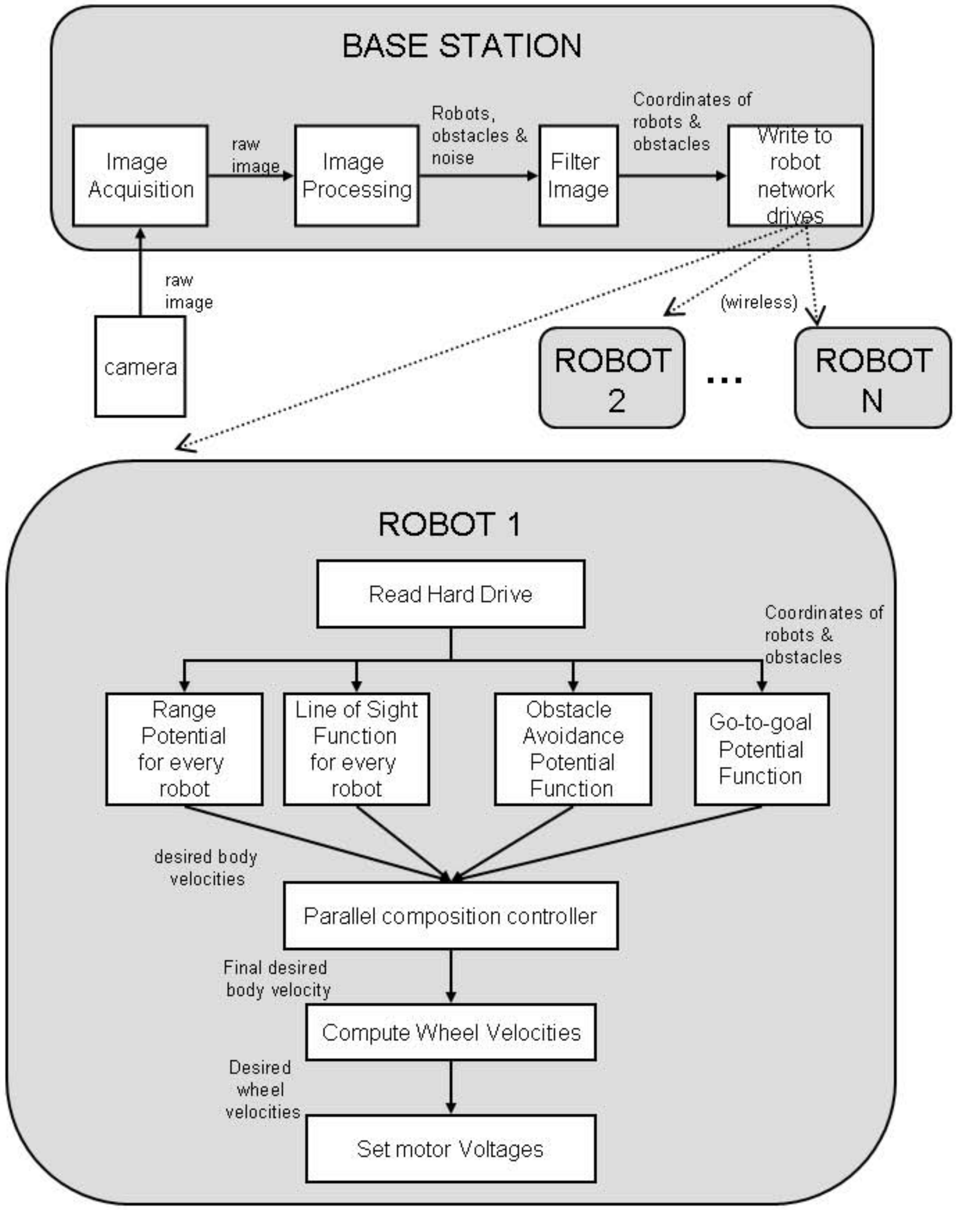


Figure 1.4: A detailed view of the steps necessary for the final experimental setup.

Chapter 2

Background

This chapter reviews some robot motion control techniques and examines the literature for related work. As reflected in Figure 1.4, robot control can be divided into three levels. A *task-level* control, which assigns a desired body velocity for a robot to complete a specific objective (e.g, avoid obstacles); a *high-level* control layer which composes various task-level velocity commands into a final desired body velocity which achieves the mission objectives; and a *low-level* controller which implements the desired velocity produced by the high-level controller, by selecting individual motor commands.

1 Task-level control

Many task level control techniques are available (see [20] for examples). However, only certain approaches are appropriate for swarm applications. A suitable task-controller should be:

- *Closed loop*: It must continually update its motion plan based on its current position, allowing it to correct for localization errors or noise that may be introduced in the course of the experiment.
- *Reactive:* It must be able to adapt to a changing environment. Each member of the swarm functions as a moving obstacle, leading to a constantly changing environment.
- *Distributed*: Each member of the swarm should be able to synthesize its own motion plan using only locally available information, without relying on a "master" robot.
- *Computationally efficient*: It must be able to recompute its motion plan in realtime, as new information becomes available. Note that the robot's on-board processing power (clockspeed of 266MHz) is significantly less than that of a desktop computer (typically 1 GHz). The speed at which motions must be computed by the robot's on-board computer is dictated by the camera's frame rate, typically greater than 5 Hz.

Artificial potential fields meet all of these criteria

1.1 Artificial potential fields

A standard and highly adaptable approach is the *artificial potential field method* [19]. The artificial potential field method involves projecting an artificial potential (*i.e.* scalar) field onto the domain in which the robot is working. If x and y are coordinates describing the robot's position the potential function is

$$\phi(x, y).$$

It is possible to define a potential field with the lowest potential value at a specific desirable location or locations and highest value at undesirable locations.

The negative gradient, defined by:

$$\dot{x} = -\frac{\partial \phi}{\partial x}$$

$$\dot{y} = -\frac{\partial \phi}{\partial y}$$

determines the desired body velocity. By moving in the direction of the negative gradient the robot will move to a lower potential in the shortest amount of time.

The distance to a goal location, the distance between robots, or the distance to the nearest obstacle can be encoded in the function itself. Figure 2.1 (b) shows a potential field for a goal located in the upper right hand corner. Part (c) of the figure shows the potential function for the robot to avoid collision with the obstacles. Part (d) of the figure shows the superposition of these two simple artificial potentials. These pictures show each potential field with the value of the field plotted as a height. This is analogous to the gravitational potential represented on a topographic map.

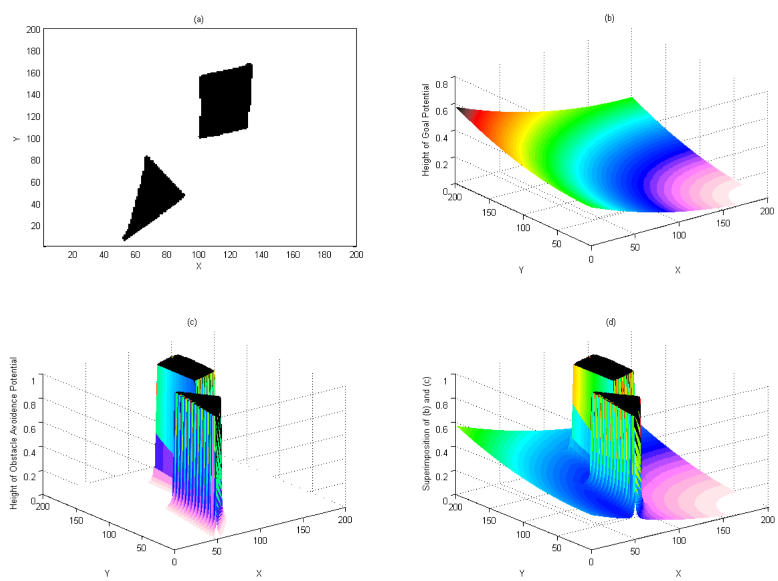


Figure 2.1: (a) Represents an overhead view of the robot's workspace with two obstacles; (b) represents the attractive potential made by the goal; (c) shows the repulsive potential produced by the obstacles; (d) is the superposition of (b) and (c). All xy axis have units of cm and the z axis has units of cm^2/s.

For many potential fields, the negative gradient can be calculated symbolically. However many other functions representing potential fields are too complex or have a discontinuity so the negative gradient cannot be evaluated symbolically. For such potential fields the negative gradient must be found using numerical methods according to the definition of the derivative

$$\frac{\partial \phi}{\partial x} \approx \frac{(\phi(x + \Delta) - \phi(x - \Delta))}{2\Delta}.$$

Through experimentation, $\Delta = .01cm$ was observed to provide an approximation sufficiently close for the engineering purposes of this project.

1.2 Basic task potentials

Several schemes exist using artificial potential fields methods for the basic tasks of obstacle avoidance and goal completion. The purpose of this project is not to investigate these ideas or functions any further. The main point of this project is to develop controllers for communication (i.e. maintaining inter–robot range and line of sight) and then combine them with pre-existing obstacle avoidance and goal completion potentials in a novel way. Therefore, standard functions as outlined in the literature are used for this project. While these are standard techniques, it is a necessary step to creating a fully functional system.

Goal completion potential functions The goal completion potential function is a shallow paraboloid. A paraboloid is a parabola rotated about the z axis to create a 3 dimensional shape, see Figure 2.2. A paraboloid is mathematically defined as:

$$\phi_{goal}(x, y) = (x - a)^2 + (y - b)^2$$

where (a, b) is the Cartesian coordinate of the goal location. As the robot closes the distance to the goal, the velocity decreases until at the goal the velocity equals zero. A paraboloid is a stabilizing function. As the robot approaches the goal, its speed decreases and thus the robot will not overshoot the goal but in theory, come to rest at the goal. [20, page 299]

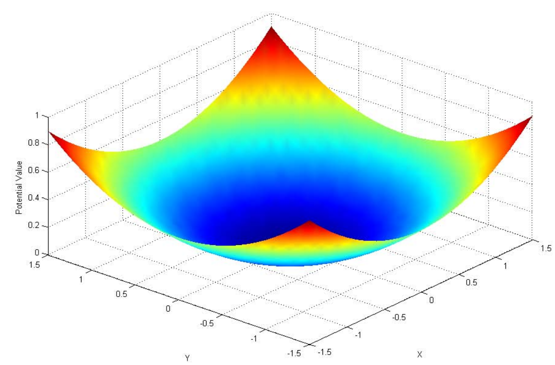


Figure 2.2: A paraboloid with the goal at the minimum. Units of xy axis are cm while z axis is cm^2/s

Obstacle avoidance potential functions Standard obstacle avoidance functions as defined by Latombe are also used [20, page 300]

$$\phi_{obst}(d(x, y)) = \begin{cases} N\left(\frac{1}{d(x,y)} - \frac{1}{D_0}\right)^2 & d(x, y) \leq D_0 \\ 0 & d(x, y) > D_0 \end{cases},$$

where $d(x, y)$ is the distance from the robot to the nearest edge of the obstacle, D_0 is an arbitrary cutoff distance where the obstacle avoidance function stops exerting influence, and N is a positive scaling factor. This function acts as a "repulsive" potential (see Figure 2.3). By changing the scaling factor and the

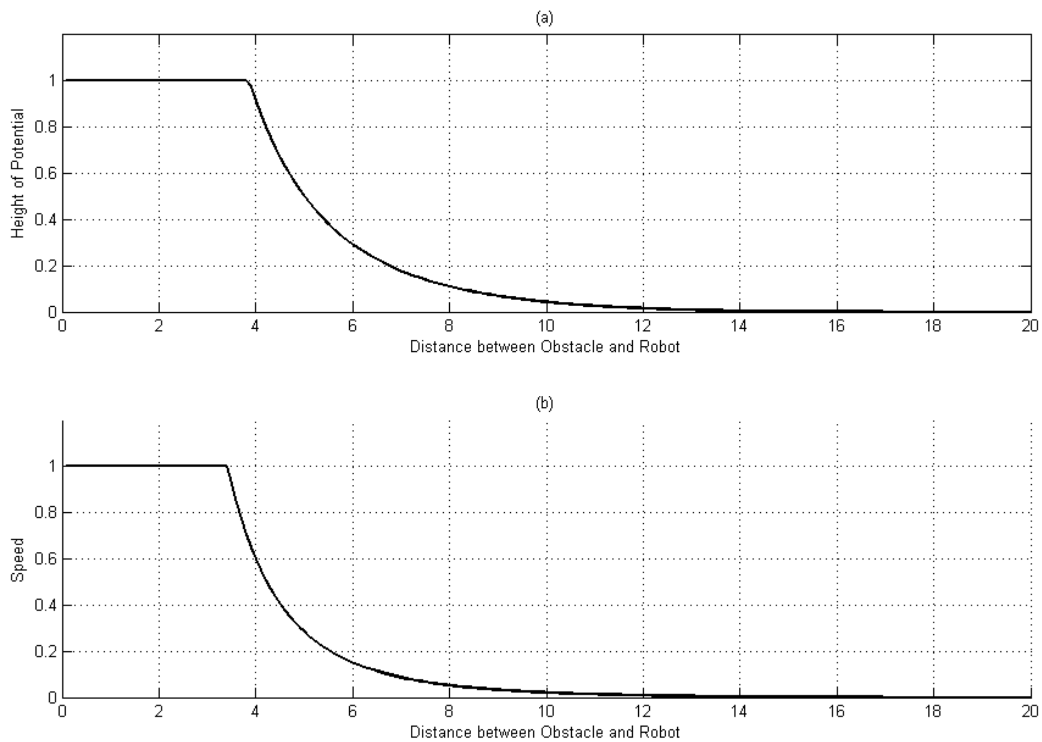


Figure 2.3: (a) A radial cross section of the obstacle avoidance potential function and (b) the speed assigned by this function. $N = 50, D_0 = 17$ and the maximum height of the potential $= 1$. x axis is in units of cm while the z axis is in cm^2/s

cutoff distance, the influence of an obstacle can be changed. The direction of the negative gradient of ϕ_{obst} (and hence velocity) is perpendicular to the nearest edge of an obstacle. As the distance increases from the obstacle the negative gradient tends toward zero, until the cutoff distance, when $-\nabla\phi_{obst} = 0$. The function's negative gradient approaches infinity as $d(x, y)$ decreases, which causes a stronger "repulsion" the closer the robot is to the obstacle, preventing a collision.

While these potential functions are standard, both of these functions were required for basic operation of a simulation and implementation. MATLAB computer code can be found in Appendix A.

1.3 Related work to communication task potentials

Little has been done with specific regard to potential functions and the communications constraints outlined above. Several papers have investigated components of communication task potentials, but none has provided a specific potential uniquely addressing the design criteria required for robust communication.

A range constraint for a swarm of robots was considered by Reif and Wang [31] and has been a component of other research such as Li and Rus [21]. Reif and Wang's work contained a potential function which had several properties in common with the Marr Wavelet and helped serve as a basis for the research regarding the range constraint. However, as the authors point out, their application of the potential function did not scale linearly with the number of members in a swarm and therefore was not appropriate for this project.

In addition, it does not allow independent specification of minimum and maximum separation distances.

Maintaining line of sight has been a component of other research ([36, 42, 3, 1]). Yet all of the approaches to maintaining line of sight rely on a leader and follower robot or only one robot moving at a time. In contrast, this project is specifically designed to be completely decentralized with no leaders or followers and each robot continuously moving. For these reasons, new controllers based upon a potential field method of control had to be designed from first principles for this project.

There is a wealth of scholarship on redundancy of a network in a computer science context ([14, 9, 23, 13]). In addition, the problem of connectedness ([41, page 161]) has been extensively studied using the mathematical field of graph theory and discrete mathematics. However little literature exists on the spatial arrangement of mobile nodes to create and maintain redundancy in a wireless network. Grabowski et al. [16] explicitly consider the creation of a redundant line of sight controller with several robots. However, the controller is based on only one mobile robot and several stationary ones and therefore not applicable to this project.

1.4 Discussion

The potential field method of control for task level objectives has several advantages over other methods of control. Potential fields are easy to visualize and there is a wealth of physical examples from which to draw inspiration and which can serve as intuitive models. Other methods of control, such as using a compensator, do not have physical analogies and cannot be easily visualized. To design a potential field, three dimensional parametric equations are written which define a potential hypothetically characterizing the desired stimulus response. To assess the potential, simulations of a robot moving according to the negative gradient of this potential field are run to determine if the actual motion meets the design specifications. This process is repeated iteratively until the desired motion is achieved.

Potential fields also have some drawbacks, which if not properly accounted for can have undesirable consequences. Most of these problems arise from the summation or superposition of multiple potentials. One problem is that potential functions can have different magnitudes or scales. For potential fields with different magnitudes, the larger magnitude potential field can overpower the smaller. A first step to engineer around this drawback is *normalization*. Normalizing the functions is the process of scaling the functions to a common, arbitrary magnitude so all the potentials will have comparable effects in composition. However, some functions or their negative gradients, for instance ϕ_{goal} are not bounded and therefore cannot be normalized. Since these functions are unbounded the velocity must be capped at a maximum value. See Figure 2.4 for an example using ϕ_{goal}; Figure 2.3 also exhibits the property of being capped. The reason for setting a maximum velocity is two-fold. First, this ensures that a function does not inadvertently overpower another function. Secondly, it is based on the physical limitations of the robot motors. The motors saturate at a certain velocity and therefore, it is not an accurate model of the physical systems if the functions can return a near infinite velocity. In addition, normalization eases the visualization of the composition of functions, as shown in Figure 2.1 (d).

A second difficulty with the summation of potential functions is that the summation of two potentials can produce undesired local minima. At a local minimum, the negative gradient is zero and thus the velocity is zero. These local minima prevent the robot from ever reaching its objective location. Local minima are always a problem with motion defined by potential functions, but it is especially difficult with a summation of potentials to not inadvertently create local minima. Even with carefully designed potential fields, in the summation of a large number of potential fields local minima can unexpectedly occur.

When dealing with a single robot, local minima are a critical design consideration. However, in swarm robotics, local minima are not as serious. This project relies on several phenomena in order to engineer around local minima. The first is a non-static potential field. In principle, for a static potential field, the potential function can be examined for these local minima. A non-static potential field continuously changes, for example, as the distances between robots change. Therefore, if a local minimum does occur, as the other

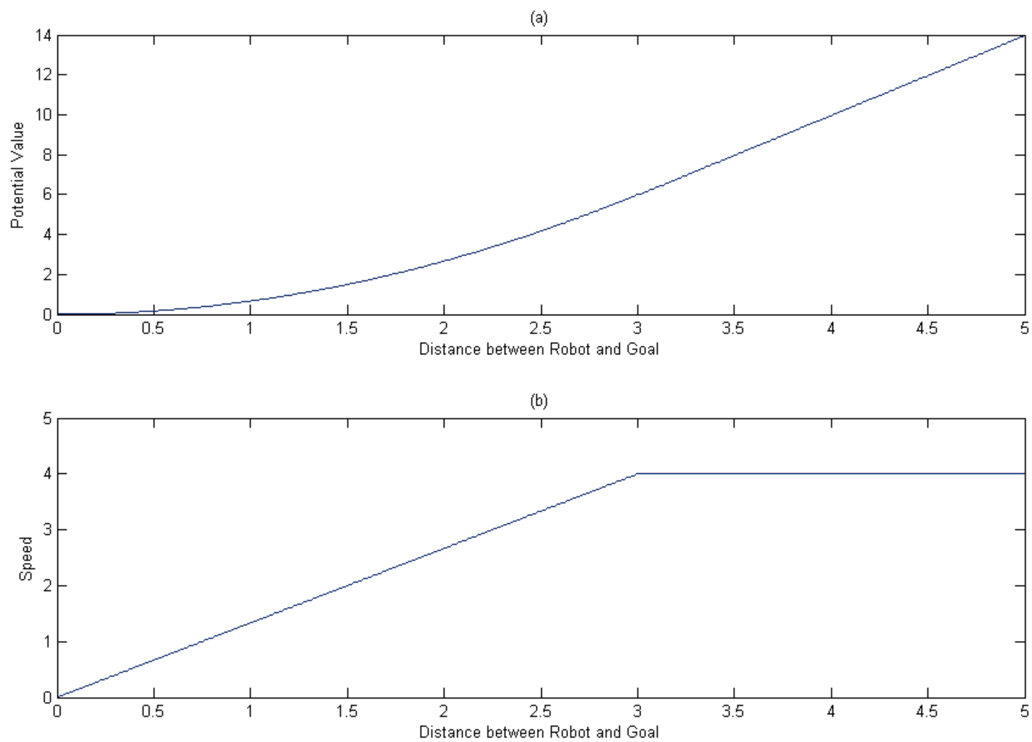


Figure 2.4: (a) A parabola that changes to a line at distance = 3 (b) the speed capped at a maximum speed of 4.

robots change position, the local minimum could subsequently disappear. Non-static potential fields have the disadvantage that they cannot be inspected and visualized as easily as static potential fields. Instead, simulations must be run in order to determine when and if a local minimum occurs. The simulations also indicate the duration of the local minimum and how seriously it affects the performance of the swarm.

The second phenomena which engineers around local minima is that swarms are designed to be fault tolerant and can lose a few of their members, e.g. to immobility, yet still accomplish the overall mission. This project intends to specifically design controllers which are able to function should the swarm lose a few of its members (Criteria 4:*Fault Tolerance*).

Finally, as will be described in Section 2.2, potential functions are not summed in a straightforward manner, reducing the number of spurious minima.

2 High-level control

After the desired body velocities are computed from each of the task potentials, a method needs to be used to compose the desired body velocities into a final velocity. A simple summation of velocities can result in undesired motions as explained above. Instead, a high level control architecture is used to combine the various body velocities into a final velocity.

2.1 Work related to high-level control

There are several control architectures used to compose task level velocity commands into a single final velocity command for the robot. A *subsumption architecture* [7] is based on a rigid prioritization of the individual task level controllers. This architecture creates algorithms which block all task-level commands from lower priorities if a higher priority is activated. This method is not well suited for this project because it forces a prioritization which will not allow the accomplishment of two disparate goals at the same time.

Another architecture is a *behavior based model* [2]. When more than one task-level command is given the velocity commands are superposed or added. This technique is designed to mimic actual biological systems. However, the straight forward addition of two or more task-level velocities does not ensure that either task is accomplished, and may create local minima.

Another control architecture is the *redundant control method* which was developed by Professor Bishop [6]. This architecture is based upon techniques used to control robot arms, and uses the extra degrees of freedom in the swarm for controlling swarm wide statistics such as the positional mean and variance of the swarm distribution. While this algorithm is designed specifically for swarms it is only used to achieve swarm-wide objectives. It cannot control the individual positions of the robots. Thus, this controller does not have any mechanism in which to ensure that the individual robots are positioned in such a way to guarantee line of sight or redundancy and therefore is not suited for this project.

Midshipman Tan [37] investigated and created a hybrid controller for a swarm of robots primarily based upon the redundant method with elements of the other two architectures. However, it too was only applicable to swarm wide objectives.

2.2 Planned approach

The planned high-level control approach is loosely based on work by Professor Esposito [11]. The basic idea is that the negative gradient direction is not the only motion direction that will decrease the potential. Any direction that consistently decreases the task potential can lead to accomplishing the objective. Therefore there is some degree of latitude in selecting the final velocity.

Essentially, the set of all velocity directions should be constructed for each task objective. One must then compute the intersection of the sets. If the intersection exists, any velocity in the intersection can achieve all the task-level goals simultaneously. If the intersection is empty, it is truly not feasible to accomplish all of these objectives at once and some must be discarded.

The method is outlined in [11] as an abstract proof. The implementation of this method is discussed in detail in Chapter 3 Section 4. Additionally, an algorithm was created for deciding which desired body velocities must be discarded in order to achieve a set of velocities for which an intersection exists.

3 Low-level control

The high-level control algorithm returns a body velocity $\dot{x}$ and $\dot{y}$ which must be changed into motor speeds so the robot can be commanded.

A differential drive controls the Koala robot's course and speed. A differential drive consists of motors attached to each set of wheels which can be controlled independently. Figure 2.5 details some critical parameters in the kinematic derivation. Let R be the wheel radii, W is the separation distance between the left and right wheels, ϕ_R and ϕ_L are the right and left wheel angles respectively. Also, let x and y be the coordinates of the point midway between the left and right wheels and θ be the heading angle of the robot as measured counter clockwise from the positive x-axis.

A difference in speed between the two motors results in the robot turning. Only the motor speeds, $\dot{\phi}_R, \dot{\phi}_L$, can be controlled so a transform from the desired course and speed to the motor speeds must be computed. Since the wheels cannot create a velocity perpendicular to their heading, instantaneous motion

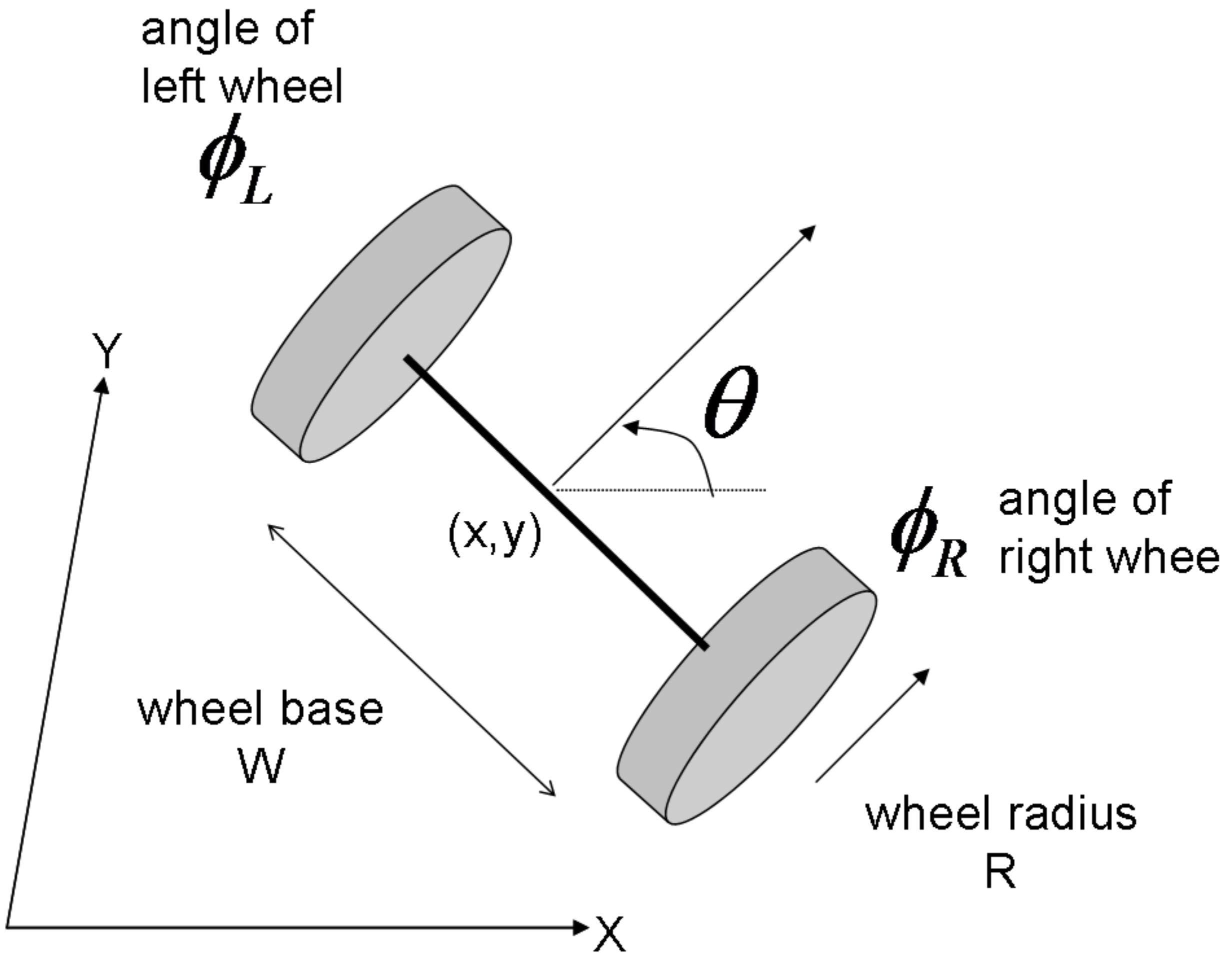


Figure 2.5: A diagram depicting a differential drive

in this direction cannot be achieved. Because of this velocity constraint, the robots are *non-holonomic*. The computer simulations used in this project constrain maximum speed but simulate a holonomic robot which can move in any direction.

A special matrix, called a *Jacobian*, transforms the motor speeds into rates of change for xy coordinates and heading, θ. Shown below is the Jacobian equation (see Figure 2.5 for definition of parameters and variables):

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = J \begin{bmatrix} \dot{\phi_R} \\ \dot{\phi_L} \end{bmatrix},$$

where J is the Jacobian matrix and is defined as follows:

$$\begin{bmatrix} \frac{R}{2}\cos(\theta) & \frac{R}{2}\cos(\theta) \\ \frac{R}{2}\sin(\theta) & \frac{R}{2}\sin(\theta) \\ \frac{R}{W} & -\frac{R}{W} \end{bmatrix}.$$

However, there are more outputs than inputs using this Jacobian matrix. This means that algebraically, J is not invertible and so one cannot solve $\dot{\phi}_R, \dot{\phi}_L$ for an arbitrary $\dot{x}, \dot{y}$ and $\dot{\theta}$. To compensate for this, a *control point* must be selected which is located a distance D away from the axle. By controlling the $\dot{x}, \dot{y}$ of this point, a new Jacobian, J_D is used to write a new equation [29].

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} \frac{R}{2}\cos(\theta) - \frac{RD}{W}\sin(\theta) & \frac{R}{2}\cos(\theta) + \frac{RD}{W}\sin(\theta) \\ \frac{R}{2}\sin(\theta) + \frac{RD}{W}\cos(\theta) & \frac{R}{2}\sin(\theta) - \frac{RD}{W}\cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{\phi}_R \\ \dot{\phi}_L \end{bmatrix}$$

Note that D must be non-zero so the matrix's columns are linearly independent. By taking the inverse of this Jacobian, wheel velocities can be calculated from the rates of change of the xy coordinates;

$$\begin{bmatrix} \dot{\phi}_R \\ \dot{\phi}_L \end{bmatrix} = \begin{bmatrix} J_D \end{bmatrix}^{-1} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}.$$

Chapter 3

Motion Control Algorithm

1 Inter–robot separation distance

There are two design specifications for a ranging potential function for this project. A wireless signal's maximum distance for effective communication, d_c, is dictated by the power of the transmitter. Therefore, the robots should remain with inter-robot distances less than d_c. The second specification is that the robots must also maintain a minimum separation distance, d_s, between each other to prevent inter-robot collision. The potential function must meet both design specifications

One candidate potential function which appears to meet both specification is the Marr Wavelet. This function is defined by:

$$\phi_{\mathrm{Marr}}(d(x,y)) = \frac{C}{2\pi\sigma}(h - \frac{d^2(x,y)}{\sigma^2}) \exp\left(\frac{-d^2(x,y)}{2\sigma^2}\right)$$

where $d(x,y)$ is the distance between the two robots, and C, h and σ are parameters defining the maximum, minimum and the dispersion of the function. Originally, this wavelet was introduced to help explain biological vision processing [24]. Though the wavelet was never intended as a potential function for robot motion control, it has many features which make it well suited for this application.

The function is set so that its maximum value is centered on the location of a target robot. When another robot evaluates this potential, the large value of the negative gradient near the center of the function at the target should inhibit the two robots from colliding. Farther from the location of the target robot, the negative gradient of the function points toward the location of the target robot and thus has the effect of pulling the robots together for communication. Under the influence of this potential, the robot will settle at the location of the minimum value of the potential. The location of the minimum value for the function represents the desired distance of separation for avoiding collision and effective communication.

Simulation makes obvious that the Marr Wavelet is not effective at bringing other robots into communication range. At large values of $d(x,y)$, the negative gradient of the Marr Wavelet approaches zero. Therefore, the evaluating robot will have almost no velocity which would bring it within communication range of the target robot.

A second function which could correct the deficiencies of the Marr Wavelet is a shallow parabola, defined by:

$$\phi_{\mathrm{sp}}(d(x,y)) = kd^2(x,y)$$

where $k \ll 1$ and $d(x,y)$ is the distance between the robots.

The magnitude of the negative gradient of a shallow parabola increases as the distance between the robots increases. Thus the shallow parabola potential function can be used to maintain range for communication even when the robots are at great distances apart. At small values of $d(x,y)$ the negative gradient approaches

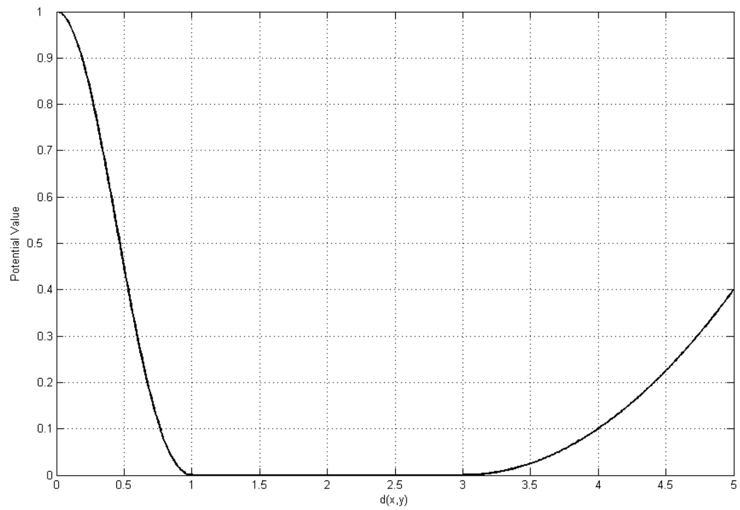


Figure 3.1: Function ϕ_{range} with $d_s = 1dm$ and $d_c = 3dm$. y axis has units of cm^2/s

zero and thus, this function does not work well for maintaining the separation distance d_s. The Marr Wavelet function works well for avoiding inter-robot collision, but fails at long distance, whereas a shallow parabola potential works well at long distances and has negligible effect at close distances.

In order to capitalize on the advantages that each of these functions present, a new function, ϕ_{range}, composed of the best of both is created. In addition, as long as the robot is between the two values of d_s and d_c, the ranging potential function should have $-\nabla\phi_{\text{range}} = 0$. Thus: (Figure 3.1)

$$\phi_{\text{range}}(d(x,y)) = \begin{cases} \phi_{\text{Marr}}(d(x,y)) & d(x,y) < d_s \\ 0 & d_s \leq d(x,y) \leq d_c \\ \phi_{\text{sp}}(d(x,y)) & d_c < d(x,y) \end{cases} .$$

MATLAB computer code can be found in Appendix B. The advantages of this function include:

- successfully meets the two design specifications of maintaining a minimum separation distance and a maximum communication distance i.e. stable minimum when $ds \leq d \leq dc$;
- separation distance and maximum communication distance can be set independently,
- easy to change values based on differing conditions or requirements;
- the function is smooth so that a derivative exists everywhere; and
- no extraneous velocity returned when robot is within specifications.

2 Line of sight

The design criteria for line of sight control is based upon the following scenario: as Robot A loses line of sight with Robot B (Fig. 3.2), Robot A should have a velocity which maintains or re-establishes line of sight. Conceptually then, for Robot A to *maintain* line of sight, the potential function should return a gentle velocity out of the occlusion zone of the obstacle with respect to Robot B. It is important to note that in the full scenario, Robot B will also be moving and evaluating an identical control law which is governing Robot A.

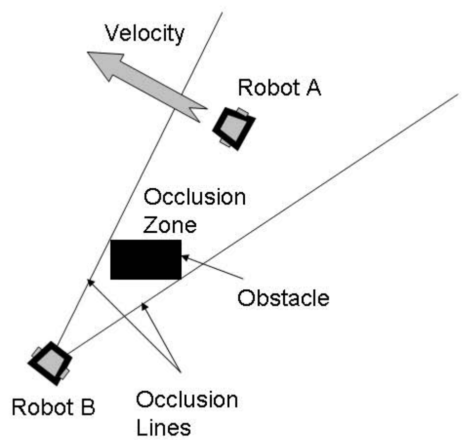


Figure 3.2: The example scenario for the line of sight problem with a stationary robot and an obstacle.

A solution to the above scenario is encoded in the potential function ϕ_{LOS}(Fig. 3.3). An upper and lower occlusion angle (θ) and a radius from the stationary robot describe the occlusion zone mathematically with polar coordinates. The only information Robot A needs to generate the potential function is Robot B's position and the shape of the obstacle. For this function, it is more instructive to define the negative gradient directly. For ϕ_{LOS}, the negative gradient will always point perpendicular to the nearest occlusion line, so that the robot exits the occlusion zone in the quickest manner possible. If $d(x, y)$ is the Euclidian distance to the nearest occlusion line (which has an angle of θ) and s is a small positive number, then for $d(x, y) + s > 0$,

$$-\nabla\phi_{\mathrm{LOS}} = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \propto \begin{bmatrix} \cos(\tan(\theta))(d(x,y)+s) \\ \sin(\tan(\theta))(d(x,y)+s) \end{bmatrix}.$$

If $d(x, y) + s \leq 0$ then $-\nabla\phi_{\mathrm{LOS}} = 0$. The offset s is included so that a small velocity is present outside of the occlusion zone to help maintain line of sight. ϕ_{LOS} is the best choice for maintaining line of sight for these reasons:

- velocity is perpendicular to occlusion lines so line of sight is re-established in shortest amount of time possible;
- a proportionate velocity to distance out of communication range; and
- an 'offset' to help maintain line of sight.

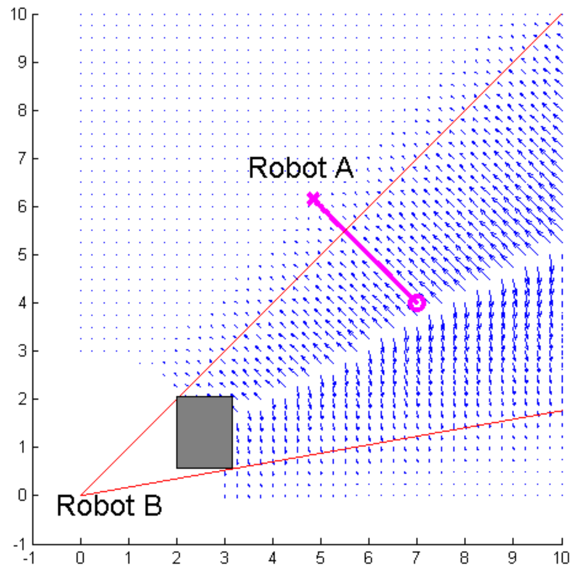


Figure 3.3: A simulation of a robot moving under ϕ_{LOS}. The velocity field created by $-\nabla\phi_{\text{LOS}}$ is marked in blue. The initial position of the robot is marked with an o and the final position is marked with an x. The red lines mark the occlusion lines. xy axes are in units of decimeters

MATLAB computer code can be found in Appendix C.

Simulations revealed that the function works extremely well for only one robot evaluating the potential function. It does not work well when both robots are evaluating and moving according to the potential function. In some scenarios, the robots will continuously rotate around an obstacle as each is moving to a different side to try to re-establish line of sight. In other cases, if the obstacle is positioned between the robots and a goal, both robots will come to rest in a local minimum created by the goal function and the line of sight potential function. This local minimum means that the robots cannot complete their goal if line of sight is to be maintained.

The solution to this problem is to emulate in the original scenario (Figure 3.2). When Robot A evaluates ϕ_{LOS} a body velocity is returned. In addition, when Robot B evaluates the potential function (since it is in the occlusion zone of A), a velocity will also be returned. However, Robot B's returned velocity can be suppressed so that the velocity equals zero. This scheme of suppressing one robot's returned velocity requires that both Robots A and B are be in agreement as to which one of them will suppress the velocity returned by ϕ_{LOS}. Suppressing a velocity does not mean that Robot B will be stationary, but rather that the desired velocity to maintain line of sight with Robot A will equal zero. All other desired body velocities from the other potential functions will still exert an influence on Robot B.

To determine which robot suppresses its returned velocity the number of communication links for each robot is evaluated. The robot which has a greater number of communication links suppresses its returned velocity. Moving around the obstacle to establish line of sight with the other robot might break other communication links and since the overall goal is to maximize the communication between robots, the greater number of communication links should be preserved. Therefore, the robot with the greater number of communication links will continue on its original course to preserve its links while the other robots maneuver to maintain line of sight. If the number of communication links is equal, then an arbitrary measure, such as the serial number is used to determine which robot suppresses the returned velocity.

It is important to note that this scheme does require a minimum of message passing between the robots. Each robot needs to pass to the other the number of communication links it has and possibly its serial number. Using a position dependant characteristic (such as which robot is closer to the goal) was considered as a method of determining which robot should suppress its returned velocity. The advantage of this method is that it does not require message passing, but only uses the information the robot already has, namely the position of the other robot. However, this type of scheme does not consider the number of communication links per robot and thus does not maximize a robust communication network (Criteria 2:*Robust Communication*).

This message passing does not violate design Criteria 3:*Distributed Operation* which states that "each robot makes its own motion control decisions based on locally available information." All information can be obtained locally by communicating with the other robot as the two robots approach a situation which could interrupt line of sight communication.

When this control law is combined with other task potentials using the communication variant of the parallel composition controller, line of sight will always maintained. Simulations confirm this result (Figure 3.4). MATLAB computer code for this parallel composition controller can be found in Appendix E and L.

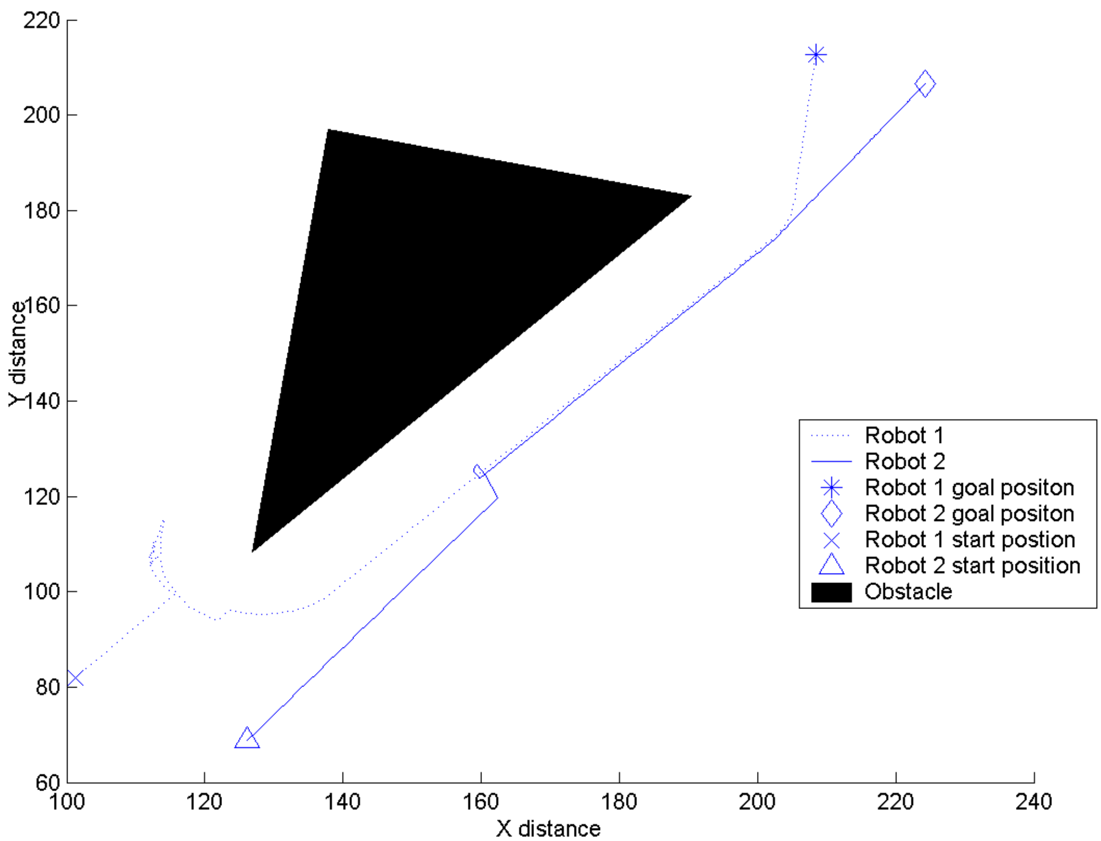


Figure 3.4: A simulation of a two robots moving according to the communication variant control law (Section 4) so that they always maintain line of sight as they move towards the goal. xy axes have units of cm.

3 Redundancy

At a minimum, a communication network is redundant if each robot maintains at least two communication links. If this condition is met, a single robot could fail, and yet all robots would still be in communication with at least one other robot. By requiring more than two connections per robot, a stronger redundancy can be created. Later in Chapter 5 the amount and degree of redundancy in a network is examined in more detail.

A first attempt at achieving redundancy was to create a potential function which would orient the network nodes in such a way as to create redundancy. After careful study and simulation, this approach was abandoned as it proved to be untenable and a more efficient method of creating redundancy was found. However, the efforts of this first attempt warrant some discussion. It was discovered that the type of functions explored cannot be scaled linearly and therefore are not good potential functions for swarm robotics.

3.1 Potential Functions for Redundancy

A first attempt at achieving redundancy was to have three robots moving toward a common point. The hypothesis is that the robot motions are coordinated in moving towards a common point, and thus would establish redundant communication expeditiously. To avoid inter-robot collisions as they approach the common point, a ranging potential function such as the Marr Wavelet would be centered on the common point to keep the robots at a safe distance from each other.

The class of geometrical constructions known as the *centers of a triangle* serve as a natural starting place for finding a common point for three robots. The locations of the robots would serve as the vertices of the triangle and a center of this triangle would be the common destination point for the robots. By carefully selecting which of the many possible centers serves as the destination point, other advantages are attained. While the main focus is still on establishing communication, the properties of a center which could be advantageous to the swarm determined which of the many possible centers were used.

The *circumcenter* is a center of a triangle which is also the center of a circle passing through the three vertices. Therefore, this center is equidistant from all vertexes. Because the distances are equal, the robots will arrive at the circumcenter simultaneously, no robot waits for the others and so it seems redundant communication should be established as quickly as possible. However, simulations with this function demonstrate that this is not a viable option for quickly establishing redundant communication. For an obtuse triangle the circumcenter lies outside of the triangle. For very obtuse triangles, the circumcenter is located a large distance from the vertices (Fig. 3.5). Thus, for an obtuse triangle, the common point of the circumcenter is a poor choice for quickly establishing redundant communication.

Another center of a triangle which has attractive properties is the *Fermat point* which is also known as the *isogonic* or *Rorricelli point*. This center minimizes the sum of the distances from the three vertices. In addition, the three angles between the lines connecting the vertexes to the Fermat point are all 120 degrees. [8] (Fig. 3.6)

The Fermat point only minimizes distances on triangles with all angles less than 120 degrees. For very obtuse triangles, the Fermat center no longer lies inside the triangle, instead the point of minimum distance is the vertex which has an angle of 120 degrees or greater.

The advantage of using the Fermat point is that it can minimize energy expended by the swarm. The farther a robot travels, the more energy is used. Minimizing the sum of the distances each robot travels also minimizes the total energy expended by the swarm. One robot may expend more energy moving to the Fermat point, but the total energy used by the system is minimized. In swarm robotics, the focus is not on individual energy expenditure, but rather on aggregate energy used by the swarm. Thus, the minimal distance point is of special interest to swarm robotics.

Simulations demonstrated that using the Fermat point as a common point works well for three robots. However, simulations also reveal that this approach cannot guarantee redundancy for swarms larger than three members. When a swarm has more than three members, the robots do not necessarily group themselves

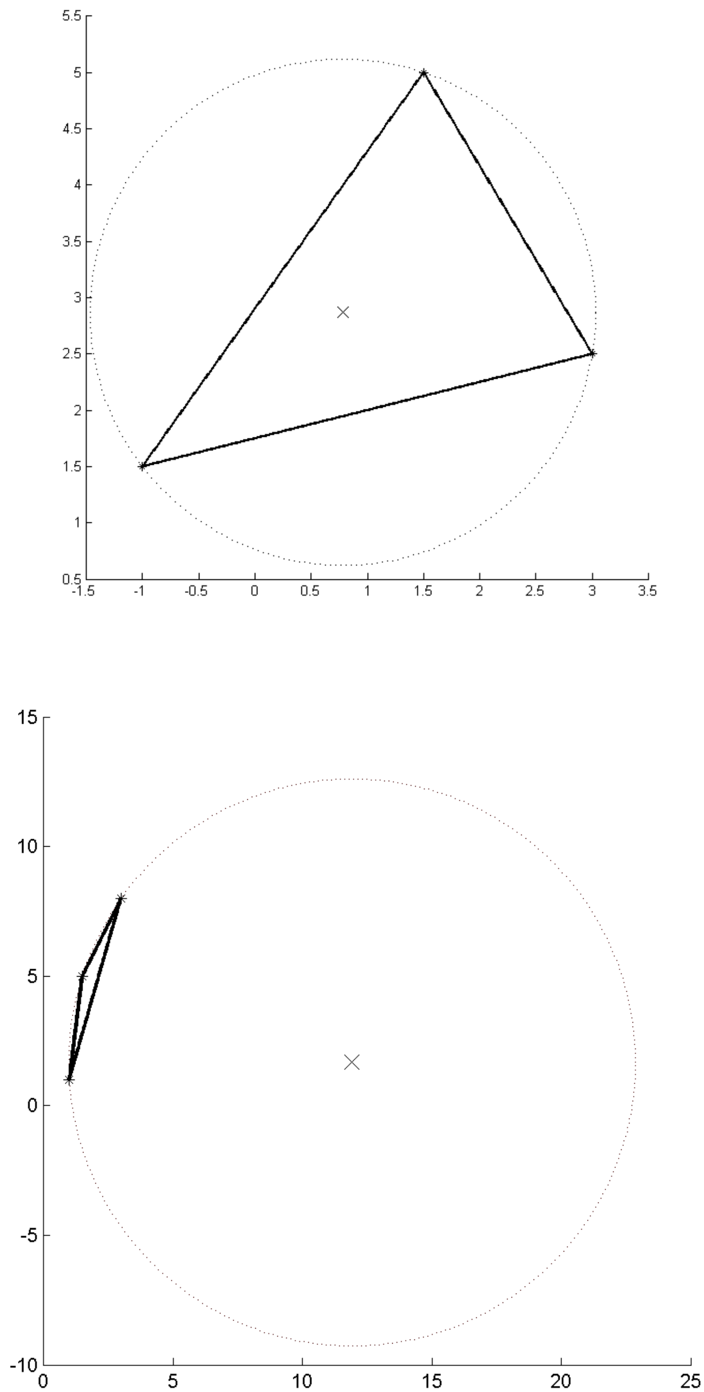


Figure 3.5: The robots are located at the triangle vertices while the "+" marks the circumcenter. As can be seen, for an obtuse triangle the circumcenter lies outside of the triangle. Note the different scale of the images.

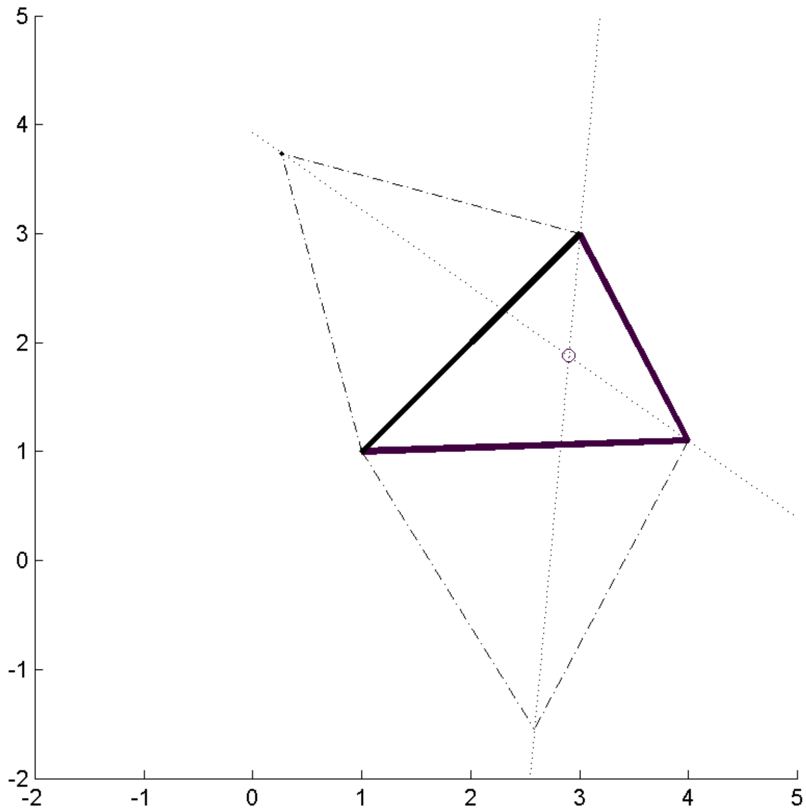


Figure 3.6: A triangle detailing how the Fermat center is constructed. The main triangle is plotted with solid lines with its Fermat center marked with a small circle. The dashed lines are used for constructing the Fermat Center.

into non-overlapping triangles. Thus, not all three robots in a triangle are moving to the same common point, causing some robots to not be in redundant communication. MATLAB computer code related to this simulation can be found in Appendix D.

The approach of using a common destination point based on centers of a triangle is appealing because other advantages can be achieved from the properties of the center. *However, unless the robots are forced into non-overlapping groups of three, this approach cannot guarantee redundancy.*

Forcing the robots into non-overlapping groups of three is not a viable option because it is not scalable. Even if the number of members in a swarm is a multiple of three, grouping the robots does not scale linearly. As the number of robots in a swarm increases, the time required to group the robots becomes unreasonable. Grouping the members intelligently, based upon their spatial arrangement, rather than on something arbitrary, like a serial number, requires an impractical amount of time. In order to group the robots successively, based upon which three have the shortest distances between them, requires

$$3\left(\frac{N!}{3!(N-3)!}+\frac{(N-3)!}{3!(N-6)!}+\frac{(N-6)!}{3!(N-9)!}+\dots\right)$$

calculations of distance where N is the number of robots in the swarm. For just 12 robots in a swarm, each robot must perform 975 calculations of distance each iteration.

A fast central processor could perform the calculations necessary to group the robots, so that less powerful hardware on board the robots is not overtaxed. Yet this violates a central principle of this project which is to keep the control decentralized. Each robot should be autonomous and not rely on calculations performed by a central processor.

3.2 Method for achieving redundancy

However, the robots do not necessarily need to be moving to a common point to achieve redundancy. Instead, as long as each robot is in communication with at least two others, then redundant communication has been achieved. Thus, a robot should be constrained in its motion so that it is attempting to maintain range and line of sight with two robots – regardless of how the other robots are moving. This means that each robot will be constrained by four velocities: a pair of range and line of sight task velocities for each of the two robots its trying maintain communication with. A new potential function is not needed, but rather a method for using these velocities as a constraint on the robot's motion. The planned high-level control method of a parallel composition controller can easily be adapted to incorporate 4 communication velocity constraints.

4 Parallel Composition Controller

4.1 Theoretical approach to composition of task velocities

The theoretical approach of the high level control is that the negative gradient direction is not the only motion direction that will take a robot to the task's goal. Any direction that consistently decreases the task potential can lead to accomplishment of the objective. Therefore there is some degree of latitude in selecting the final velocity. Each task velocity constrains the final velocity $(\dot{x}, \dot{y})$ to be such that:

$$-\left[\frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y}\right]\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} = \dot{\phi} \leq 0.$$

Geometrically this means that at any point, a velocity within $\pm 90°$ from the negative gradient will decrease the task potential (Figure 3.7). Thus the desired body velocities are constrained to a $180°$ arc. By exploiting this latitude for several vectors, it can be possible to find one single vector that will consistently decrease *all* of the task potentials and thus, one direction will meet several of the goals simultaneously.

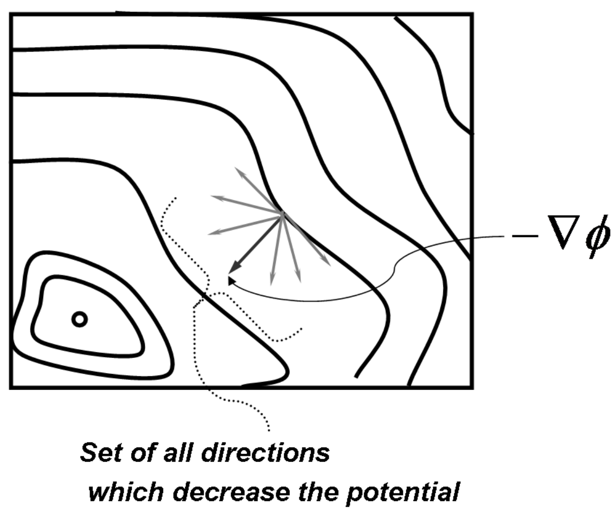


Figure 3.7: Any vector within $90°$ of the negative gradient will decrease the potential function.

However, not all sets of vectors can be reduced to one composite velocity which decreases all potentials. Thus, it must be determined if a set of task vectors are feasible. In order to be feasible, all vectors must lie on the same half plane which is determined by satisfying the following statement:

$$(\exists V_i)|(\forall V_j)[(V_i \times V_j) \geq 0 \vee (V_i \times V_j) \leq 0].$$

If no V_i exists, then the algorithm described in the next section is used to discard some of the velocities until a feasible set of vectors is constructed.

Geometric speaking the above equation is equivalent to taking the intersection of all $180°$ velocity constraint arcs (Figure 3.8) and determining if the intersection is non–empty. A non–empty intersection is the cone of feasible velocities which can decrease all of the potentials. If the intersection of the arcs is empty, then again, some of the task potentials must be discarded.

The problem of determining if a set of 2-D vectors is feasible simplifies to examining if there exists an exterior angle which is greater than $180°$ (Figure 3.9). If this is the case, then the intersection will be

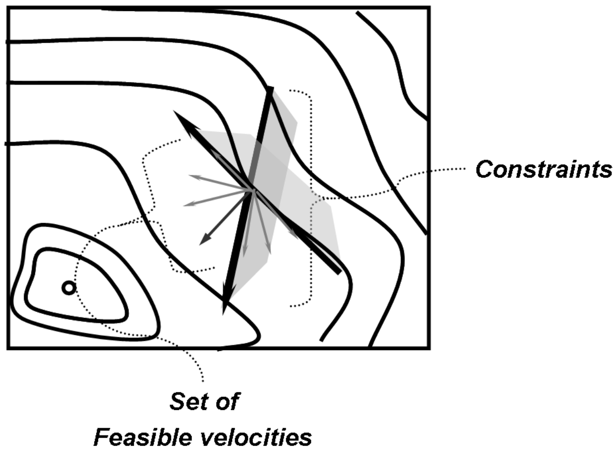


Figure 3.8: The intersection of all constraints determines the set of feasible velocity directions.

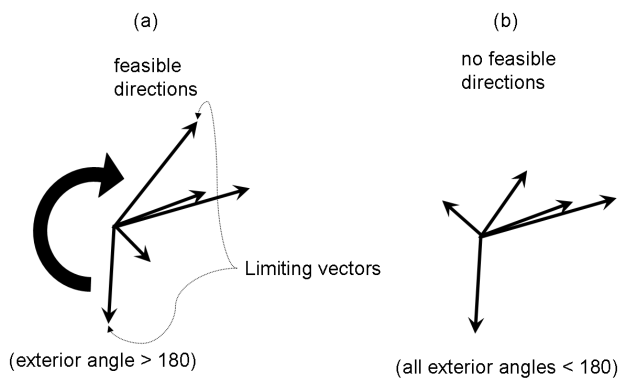


Figure 3.9: (a) a set of vectors for which it is feasible to compose into a final velocity to decrease all potentials. (b) a set of vectors with no such feasible direction, as determined by the exterior angles.

non-empty and a velocity can be found which decreases all potentials. The two vectors which make up this exterior angle are the *limiting vectors* and are used to construct the composite velocity.

Given a set of feasible velocities a final composite velocity is composed based on the ideas of Esposito and Kumar [11]. However, Esposito and Kumar only consider a *direction field* (i.e. only unit vectors) rather than a velocity field. Secondly, the authors assign a rigid prioritization of only two directions to compose a final velocity.

There can be any number of vectors in the final set of feasible vectors but only two vectors from this set, the limiting vectors, determine which direction will be the final velocity. The inside orthogonal vectors to the limiting vector are labeled *bias vectors* (Figure 3.10). These vectors represent the maximum deviation from the limiting vectors which will still decrease that task potential. Thus, the arc between the bias vectors defines an arc of velocity directions which will simultaneously decrease all potentials. Next, within this arc, a direction and magnitude for the composite velocity must be chosen. To accomplish this, the magnitudes of the limiting vectors are used with the direction of the closest bias vector (Figure 3.11). The resulting vectors are then summed using vector addition. By using vector addition, the magnitudes, and therefore, the urgency of the task velocities are reflected in both the magnitude and the direction of the final velocity.

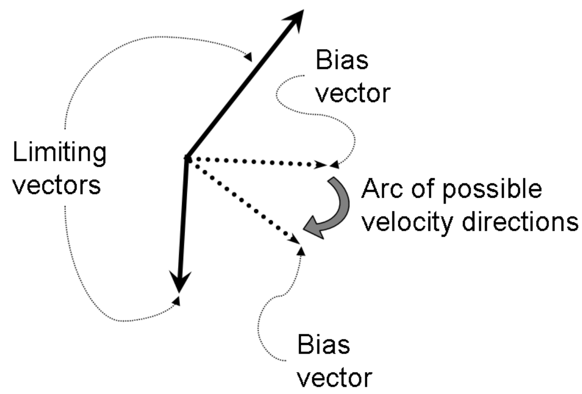


Figure 3.10: The arc of feasible directions are determined by the angle defined by the bias vectors.

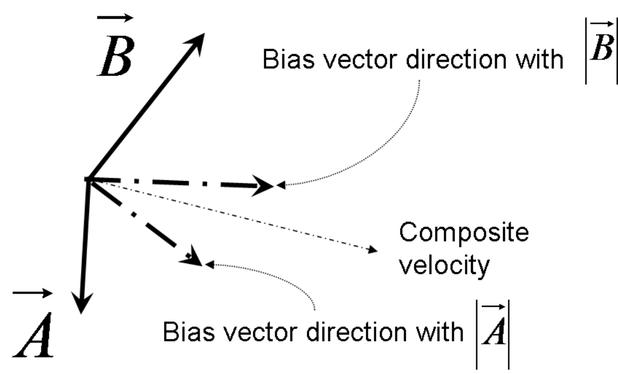


Figure 3.11: The composition of the final velocity based on the magnitudes of the nearest limiting vectors and $\pm 90°$ direction from the more distant limiting vector.

A special case of this composition scheme is when the limiting vectors are less than $90°$ apart. In this cases, both limiting vectors will lie within the arc of feasible directions (Figure 3.12). Therefore, vector addition of the limiting vectors can be used to compose a final velocity.

4.2 Algorithm for finding a feasible set of vectors

Should the set of task potentials not result in a feasible set of vectors, an algorithm is needed for determining how to compose a subset of vectors which is feasible. This algorithm is illustrated in Algorithm 1.

At each iteration a robot evaluates the potential functions for the go-to-goal function and obstacle avoidance which each return a suggested velocity (or velocities in the case of multiple obstacles). In addition, a robot individually evaluates the range and line of sight potential functions with every other robot. These potentials return a suggested velocity LOS_i and R_i for the i^{th} robot. The suggested velocity is based on the negative gradient as descried in Chapter 2 Section 1. From this set of suggested velocities a feasible set of vectors must be found so a final velocity can be composed in the manner described above.

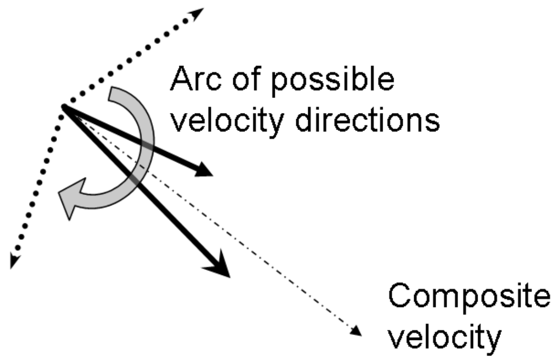


Figure 3.12: The composition of the final velocity for the case when the limiting vectors are less than 90° apart.

Algorithm 1 The Algorithm for generating a feasible set of vectors

$P \leftarrow$ set of high priority vectors
$C_{sum} \leftarrow [(LOS_1 + R_1), (LOS_2 + R_2), (LOS_3 + R_3), \dots]$
while P is infeasible **do**
  $P = P - \min(P)$
  evaluate P for feasibility
end while
$\bar{A} = P$
$M \leftarrow$ number of desired communication links
$Q \leftarrow$ number of actual communication links
for $(M - Q)$ **do**
  $\exists(i)[(LOS_i + R_i) = \min(C_{sum})]$
  if $LOS_i \cup \bar{A}$ is feasible **then**
    $\bar{A} = LOS_i \cup \bar{A}$;
  end if
  if $R_i \cup \bar{A}$ is feasible **then**
    $\bar{A} = R_i \cup \bar{A}$;
  end if
  $C_{sum} = C_{sum} - (LOS_i + R_i)$
end for

Given a set of infeasible vectors, the first step to finding a feasible subset is to group the vectors based on a loose prioritization scheme. The high priority signals include: go-to-goal, obstacle avoidance and maintaining the minimum separation distance between robots. The lower priority signals are all of the communication vectors.

The high priority signals are checked to see if by themselves they can be composed into a feasible velocity. If so, these vectors are labeled $\bar{A}$ – the set of vectors which will eventually be composed into a final velocity. If not, then the high priority velocity with the smallest magnitude is discarded. The potential functions are defined in such a way so that the larger a velocity's magnitude is, the higher the urgency. Thus, discarding the smallest vector is discarding the least urgent velocity. At the next iteration, all potential functions will be evaluated and so this velocity is only discarded for one iteration. This process of discarding vectors continues until a feasible set of high priority signals is found which is labeled $\bar{A}$. A feasible subset is guaranteed to be found because at a worst case, there will be two vectors and for all sets of two vectors a feasible velocity can be found. [1]

Next, the communication subroutine is used to find those communication velocities, LOS_i and R_i, such that the union with $\bar{A}$ is still a set of feasible vectors. The actual number of communication links per robot, Q, is compared against M, the desired number of communication links per robot. The communication subroutine seeks to find velocities which will put the robot in communication with $M-Q$ more robots. Thus by following these velocities eventually $M = Q$.

This project requires that both range and line of sight be attained before communication is established. Thus, both range and line of sight velocities for the i^{th} robot are evaluated for feasibility together. The pair of communication vectors which require the least amount of energy (i.e. the smallest magnitude of $LOS_i + R_i$) are selected first. If either LOS_i or R_i are feasible with the current $\bar{A}$, the vector(s) is/are appended to $\bar{A}$ and a counter is incremented by 1. Successive feasibility tests of LOS_i and R_i with $\bar{A}$ are conducted until the counter is equal to $M - Q$ or no more vectors are left to be tested.

However, LOS_i is tested for feasibility first. This induces a slight bias towards completing line of sight before the range constraint. While LOS_i may be feasible with $\bar{A}$, R_i may not be feasible with the set of vectors $LOS_i \cup \bar{A}$. If obstacles are assumed to block communications (such as a dense wall) then decreasing the range without being in line of sight will not help establish communications, the robots must first be line of sight before being in range will establish communication. If LOS_i is not feasible then R_i is still examined to see if it is feasible with the current $\bar{A}$.

When the counter equals $M - Q$ or no more communication pairs remain to be tested, $\bar{A}$ is composed into a final velocity in the manner described in the previous section. This algorithm achieves the goal of meeting the dual, and sometimes conflicting requirements of executing a primary mission while at the same time establishing a robust communication network.

4.3 Communication Variant

In addition to the above algorithm, a communication variant was tested to determine its effect on creating a robust network. The algorithm is exactly the same, except that the go-to-goal function is no longer considered a high priority. Instead, there are three classes of priorities. Class one consists of obstacle avoidance and maintaining minimum separation, essentially the requirements for safe navigation of the robots. Class two consists of the communication vectors and the third class only contains the go-to-goal function.

The algorithm proceeds in the same manor described above for the first two classes. Once an $\bar{A}$ has been determined, the go-to-goal suggested velocity is checked for feasibility with this $\bar{A}$. If feasible, this suggested velocity is amended to $\bar{A}$ and the final composite velocity is calculated. However, if it is not feasible, $\bar{A}$ is immediately calculated into a final velocity.

The motivation for testing this controller was to see the effect of prioritizing communication over the go-to-goal function on the overall effectiveness of the swarm in completing its dual missions. Secondly, it was motivated to develop a controller which could guarantee that line of sight could be achieved. While LOS_i by itself would establish line of sight, it is often discarded by the first controller. The line of sight potential is evaluated when the robots are around an obstacle. In these cases usually the obstacle avoidance function is returning a non-zero vector. Thus the high priority vectors of the go-to-goal and obstacle avoidance are activated and almost always 'lock out' the line of sight vector from being feasible. By changing the go-to-goal function to be a lower priority than the line of sight velocities, the go-to-goal function will be infeasible and thus discarded. Simulations demonstrate that using this controller will always guarantee line of sight. Simulations verify this result, (Figure 3.13). MATLAB computer code for both controllers can be found in Appendix E.

[1] The worst case is two vectors exactly 180° apart. For this case, there are two vectors (±90° of the vectors) which neither decreases nor increases the potentials. None the less, in the worst case the vectors can be composed into a velocity

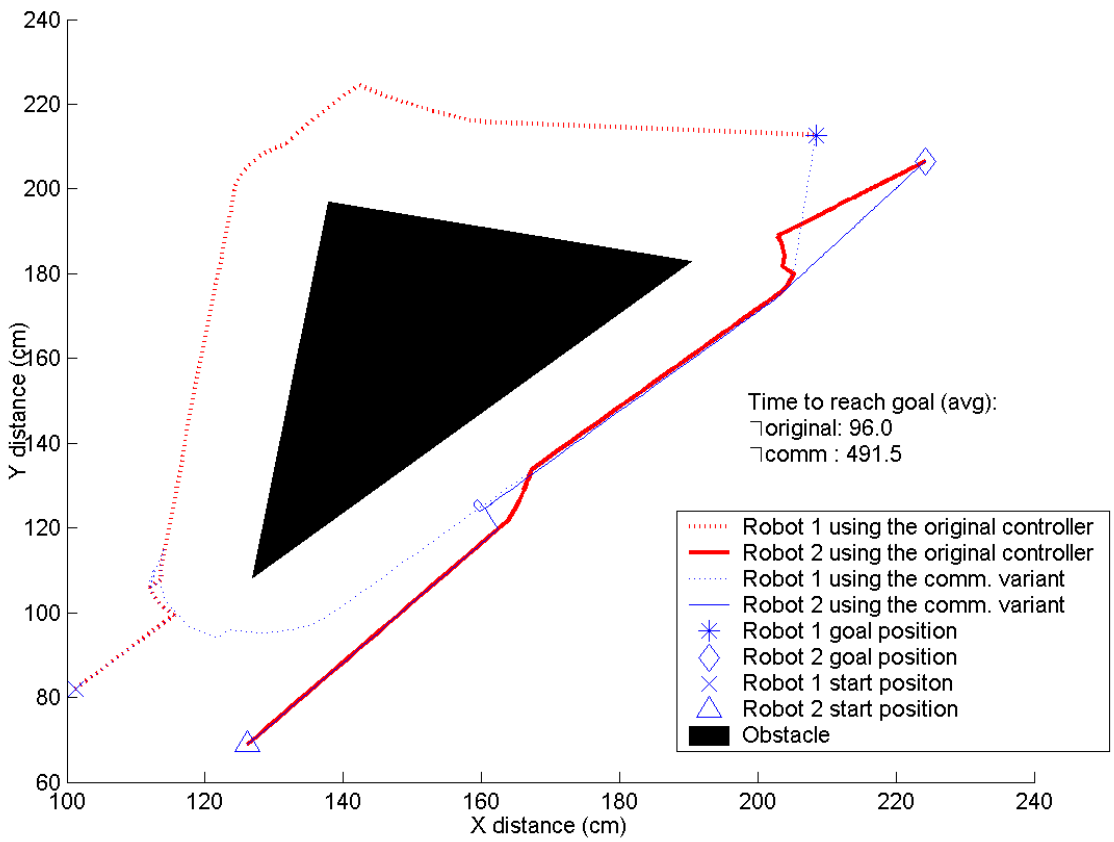


Figure 3.13: Two simulations of a two robots moving according to either the original or the communication variant of the controller. The starting and ending positions of the robots are the same in each simulation.

Chapter 4

Experimental Setup

Simulations provide much useful information, and serve as a flexible test bed for examining several different scenarios. A simulated environment can be easily rearranged to provide several different configurations of obstacles and several different numbers of robots. However, simulations do not provide the experience and insight of testing the controller in the real world. A second phase of testing and experimentation of the controller is to implement the controller on the Koala robots.

The Weapons and Systems Engineering Department purchased eight Koala robots (see Figure 1.2) from the Swiss company K-team. For the most part, these robots have not been used in any projects, experiments or research. No one in the department had any experience programming or working with them. The robots came with very little documentation and instructions. The operation and configuration of the robots had to be deduced through experimentation and testing.

Several distinct challenges had to be overcome before the robots could be used for demonstrations, testing and experimentation. The basic experimental set up is shown in Figure 4.1 with a detailed block diagram in figure Figure 4.2. Whenever possible, standard techniques were used for the experimental setup so that a majority of the research time could be spent on researching and evaluating the motion control algorithm. However, project specific implementations had to be designed and some non-standard techniques were used.

1 Localization via computer vision

The first challenge to creating the necessary experimental setup was the problem of localization, i.e. a robot knowing its position in relation to the environment. Localization is a principal challenge in any mobile robotics experiment. The purpose of this project is to investigate a novel controller to accomplish the two disparate goals of robust communication and primary goal completion, not to investigate a novel technique for the problem of localization. Providing a solution to the problem of localization was a necessary step for creating an experimental setup which could test and investigate the novel controller.

The solution designed for this project was to employ a single ceiling mounted overhead camera to reproduce the position information which could be gained from GPS receivers or local sensors. Reproducing position information in this manor is a common approach and allows for the complex problem of localization to be dealt with using the standard techniques of computer vision. The design of the computer vision system required interfacing a camera with the MATLAB environment so image processing could be facilitated. In addition, a scheme for identifying and orienting each robot had to be designed.

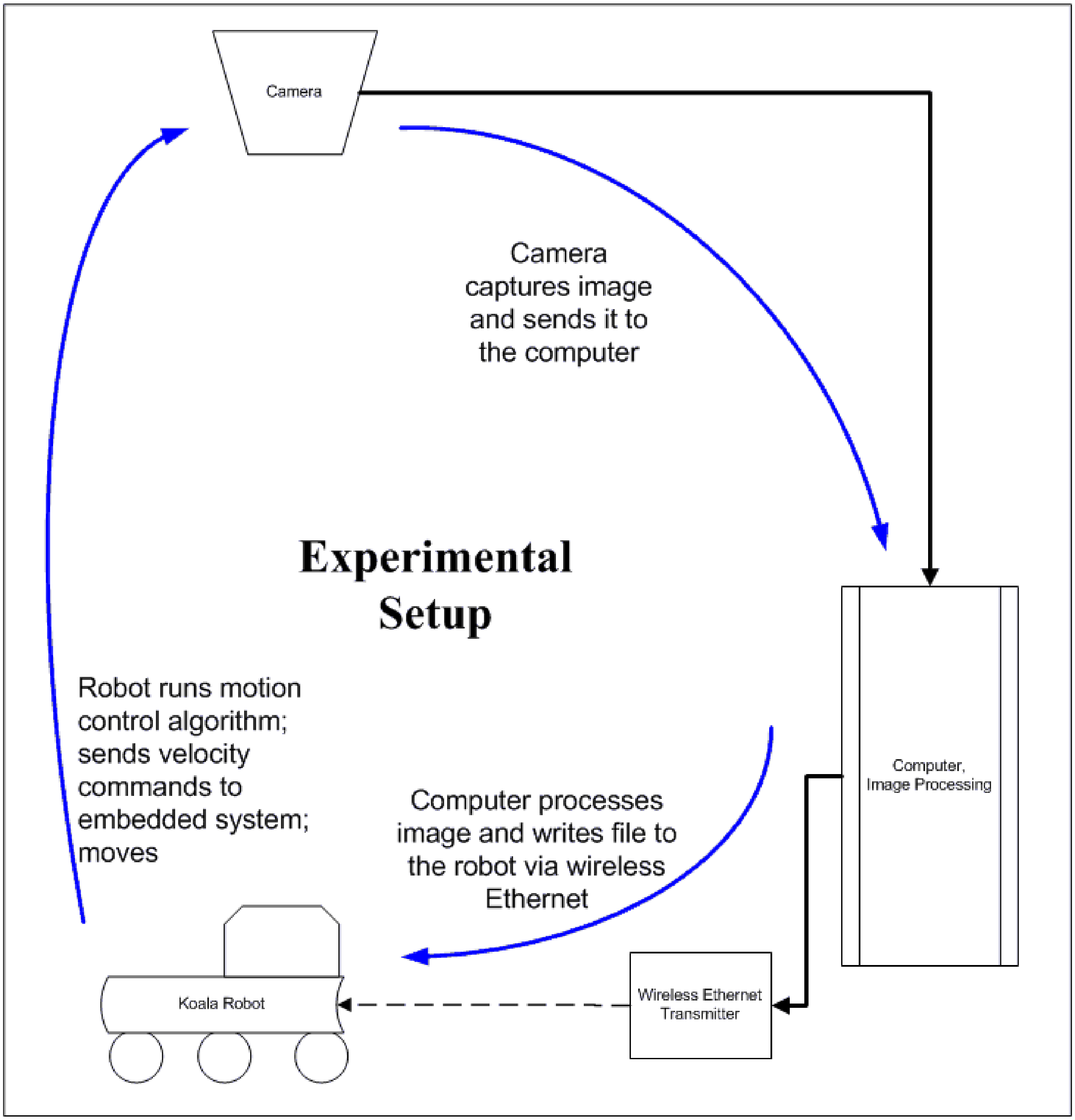


Figure 4.1: A diagram of the overall experimental setup.

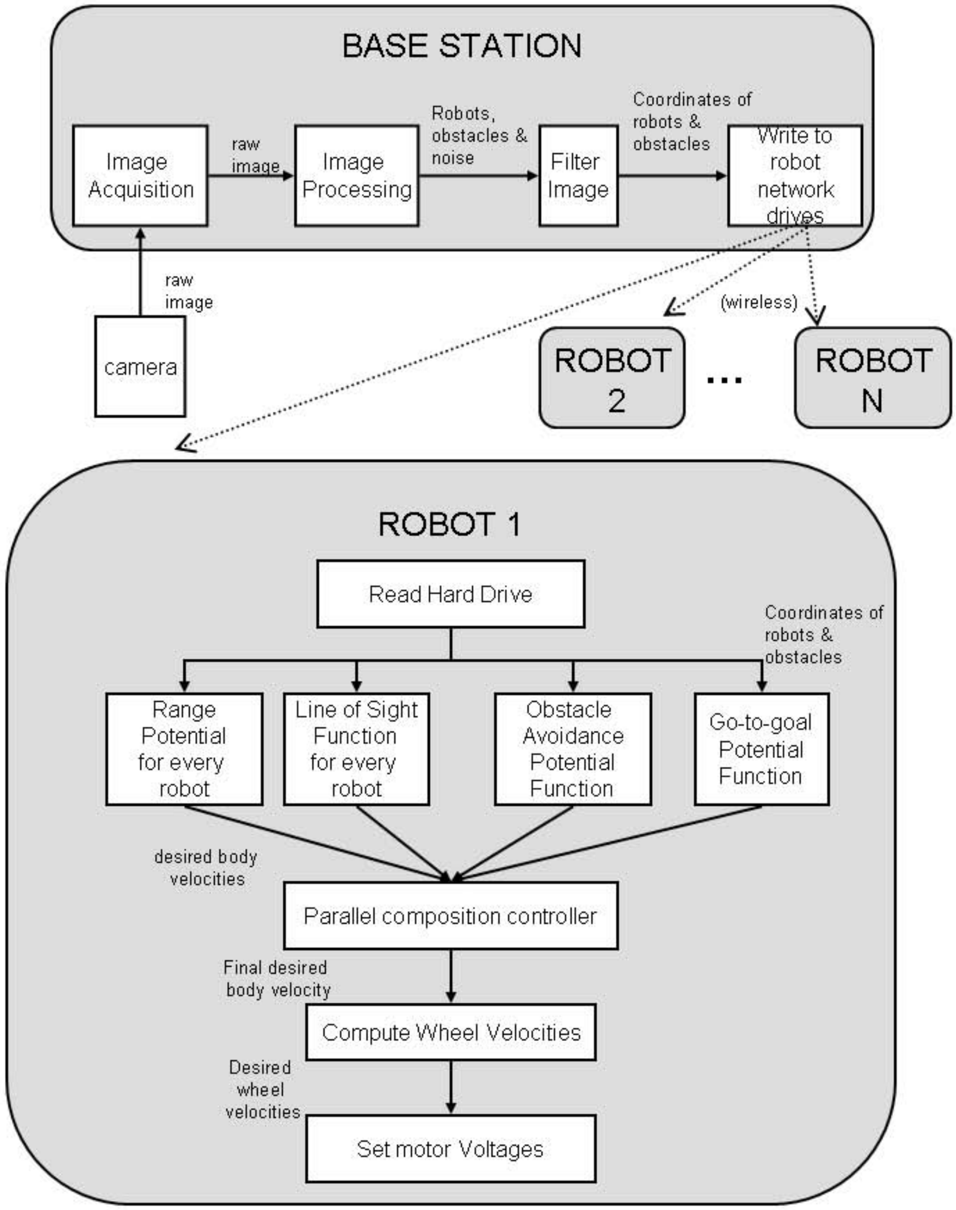


Figure 4.2: A detailed view of the steps necessary for the final experimental setup.

1.1 Computer vision hardware

The camera used in the vision system is a *FireFly2* from Point Grey Research. This camera was selected because it can provide a high resolution (640×480 pixels) color image in standard formats for image processing (See Appendix F for technical specifications). Using the image acquisition toolbox for the MATLAB high level mathematical programming language in conjunction with the IEEE 1394 device driver, a MATLAB program is able to capture an image which can be easily used for image processing. The camera is hung from the ceiling using a mounting bracket created by George Burton in the Machine Shop, see Figure 4.3. This bracket allows the camera to be hung from the drop-ceiling struts in a secure manner so that the lens is parallel to the floor. This maximizes the amount of workspace area the camera can observe. The overhead position also allows the camera an un-obstructed and non-interfering view of the workspace. The resolution provided by the camera is 1pixel/cm^2 with an overall workspace area of $\approx 7.7m^2$.

Figure 4.3: The mounting bracket.

1.2 Image processing

The identification scheme designed for the robots consists of tagging each robot with two blocks of the same unique color, a large block on the aft end and a smaller block on the forward end. The vision system must be able to differentiate between the different robot colors and background color. *Color segmentation* is an image processing technique which separates and sorts the colors in an image. Most digital images contain three layers of data: a red, green and blue layer[1]. Each color layer is an array of data which contains the intensity of that color at every pixel. When the three layers are superimposed, a full color image is created.

[1]In the actual implementation, the YUV (Luminance, Blue Chrominance, Red Chrominance) color layer scheme was used. This color scheme was selected because when the camera used this scheme it captures less pixels per image and thus allowed for a faster processing time. Though the image resolution is reduced, this did not significantly effect the accuracy of the localization.

Segmenting an image involves changing a color intensity layer or layers, which can have values ranging from 0 to 255 to a binary image which only contains values of 0 and 1. Once an image has been segmented to a binary image, several properties can be extracted by image processing algorithms.

For instance, creating a binary image of only purple pixels would require finding all pixels which have a red intensity > 200 and also have a blue intensity > 200. All those pixels which meet the purple criteria are set to 1 while all those which fail are set to 0. A binary image is created for each color of interest. The vision system needs to be calibrated for each color of interest and the lighting conditions of the workspace. Appendix H contains a table of the color layer intensity values used to segment each of the colors of interest.

Within a binary image an object is defined as a contiguous region of pixels which have a value of 1. Because noise in a binary image consists of objects which have a small area, noise can be effectively filtered by only using those objects with relatively large areas.

A filtered binary image should only contain two objects which correspond to the large and small blocks of color on the robot. As long as the objects' areas are significantly different, they can be processed to determine the robot's position and orientation. The property of an object's *centroid* is analogous to a two dimensional object's center of mass. Thus, this property is used for finding the center location of an object. Since the larger block is roughly centered on the robot, its centroid is used as the position of the robot in the workspace. The smaller block's centroid is used as the control point for low level control (see Chapter 2 Section 3). In addition, a vector from the centroid of the larger block to the smaller block determines the orientation of the robot.

In addition to the identifying the robots, the vision system also determined the location and edges of the obstacles. Since the obstacles are stationary, the vision system only needs to determine their location at the start of the experiment. In this project the obstacles were green in color to separate them from the robots.

By profiling and streamlining the MATLAB programs (Appendix G), and rewriting some of the image processing library functions, the base station was able to identify and orient the robots in $< .1$ seconds. This speed is fast enough to allow for real time updates of the robot's position.

2 Communication between the base station and robot

The next challenge in the experimental implementation was to develop a method for transferring data between the base station and each robot. The information extracted from the image processing programs needed to be transmitted through a wireless network to the robot's hard drive so that the robot could update its motion based on the new position information gleaned from the camera.

Each of the robot's PCs have a wireless Ethernet card which serves as the physical layer of a network architecture. In order to transfer the files from the base station to the robots, file transfer protocol (FTP) or HTML-based approaches were considered. Ultimately, a mapped network drive of the hard drive of the robot's PC onto the base station was the appropriate solution. Using this approach, the image processing or motion control programs could write the extracted data *directly to the robot's PC's hard drive.* This approach is easier to integrate with a control program written in the C programming language and also provided a quicker means of transferring information between the base station and robot than an FTP or HTML approach.

The base station, which uses the Windows operating system, requires that the software program *Samba* be running on the Linux PCs. Samba uses the TCP/IP protocols to allow a Windows PC to interact with a Linux PC as if the Linux PC was a Windows file server [39]. By installing Samba on the Linux PC, the base station, acting as a host, was able to access the hard drives of the Linux PCs as seamlessly as if the Linux hard drives were on a Windows machine.

Another approach considered was converting the base station into a Linux machine. However, this would require reformatting the base stations hard drive, installing and configuring Linux, installing MATLAB for Linux and finally networking the Linux base station and robot PCs. Linux is an open source operating system without standardization of user installation and configuration. Linux is purposefully non-centralized, therefore many of the components of the operating system must be downloaded, installed and configured individually. There are few commercially available Linux products and software available. Therefore, open-source software, with scant documentation must be used. Without substantial experience, these factors make it difficult to install and correctly configure a Linux operating system for a specified application as required by this project. Even if all of the above steps had been completed in a satisfactory and timely manner, it is doubtful that the special device drivers necessary for the overhead camera used in the project would run under Linux.

3 Programming and control of Koala Robots

The Koala robots have two on-board computers. The first is an embedded system which consists of a small amount of flash ROM, a Motorola 68K micro-processor and various robot hardware such as motor control circuits, sensors, analog to digital converters, etc. The Motorola 68K micro-processor controls the motors, sensors, and internal operation of the robot. The flash ROM can be loaded with a user's programs. However, the programs on the flash ROM must be written in a highly specific robot language. In order to write a program in the robot language a cross-compiler is need to change a program written in the C computer language into the robot language. [5]

The second computer that is on board the robot is a standard Pentium, 266 MHz personal computer. The computer uses a 20GB hard drive and a wireless Ethernet card. This computer uses laptop-like components that are much smaller than their desktop counterparts so the robot can easily carry the computer. With an externally attached monitor, keyboard and mouse, this becomes a fully functional PC running the Red Hat Linux Operating System version 7.1. Linux was preloaded on the computer. Because of the limited processing ability of the computer, Linux is used because the operating system overhead can be easily limited. The two computers on the robot are only connected together by a serial line.

There are two ways for the user to control the robot. The first is a *compiled* method of control where a program is written and cross-compiled on the Linux PC. The program is then downloaded via the serial line to the robot's flash ROM. The robot then executes this program until the program ends or the robot is shut off, which ever comes first. A second way for the user to control the robot is an *interpreter* approach. Simple robot language commands are typed into the Linux PC and then sent directly down the serial line for the robot to execute. Only one command can be sent at a time and must be in the highly specific robot language. In addition, convenient programming constructions such as loops and if-then statements cannot be used.

Since the interpretive method of control does not allow the robots to be autonomous, but rather reliant on a user typing commands, the compiled approach must control the robots. This requires the installation a cross-compiler onto the Linux PC. The cross-compiler allows a user to write programs in the standard C programming language rather than the hardware specific and tedious robot programming language. The cross-compiler then translates the C code into the robot programming language which only then can be downloaded and executed on the embedded system.

In addition, a method was needed which could allow the Linux PC to communicate to the embedded system using the serial line. The embedded system has several pre-set modes and speeds of serial line communication. However, two way communication between the PC and the embedded system could not be established. Therefore, compiled programs running on the embedded system could not be debugged and troubleshot in an efficient manor.

An investigation of two way communication on the serial line was undertaken using the interpreter control method because it provided instant feedback. When a user types a command into the Linux PC, the embedded system executes the command and sends a response. At this point, a better use of the computers for control was designed. A small C program could be written which would run on the Linux PC and mimic a user typing in commands. This approach streamlines the process of transferring data from the base station to the embedded system. The Linux PC would act as a relay between the base station and the embedded system. Without the Linux PC acting as a relay, the embedded system would have no way to receive the new information which would be coming from the base station.

Since the Linux PC does not have MATLAB and given the processing power, it would be inefficient at executing it, all the potential functions, controllers and algorithms would need to be converted and compiled into the C programming language. Converting these functions and controllers also meant that any subsequent design or modification of the controllers tested and modeled in MATLAB would require a new conversion and compilation into the C programming language.

However, a more time effective method of implementing the controller was designed. The brute strength of the base station's processing power was used to calculate the control algorithm for each of the robots in the experiment. The base station could run MATLAB efficiently and therefore the programs did not need to be converted to the C programming language. The base station was used to calculate the wheel velocities of the each of the robots. It is important to note, that this system of implementing the control system does not violate a central premiss of the project which is to keep control decentralized (Criteria 3:*Distributed Operation*). While the base station does all of the calculations, it does each calculation *based only on the information each robot would have*. Allowing the base station to do the calculations is merely an efficient use of the computing power available. With more powerful processors on board the robots, or time to convert the programs to C, there is absolutely no reason why the control calculations could not be performed on board the robots. Appendix J contains the MATLAB computer code for these sections.

The robots themselves still run a small C program running on the Linux PC to collect and transmit the calculated wheel velocities. Once the base station had finished calculating the wheel velocities for a robot, these values are written to a file on the robot hard drive (as discussed in Section 2). The small C program running on the Linux PC reads this file and writes the values down the serial line to the embedded system. The embedded system takes the wheel velocity values and translates them into motor voltages (Figure 4.4).

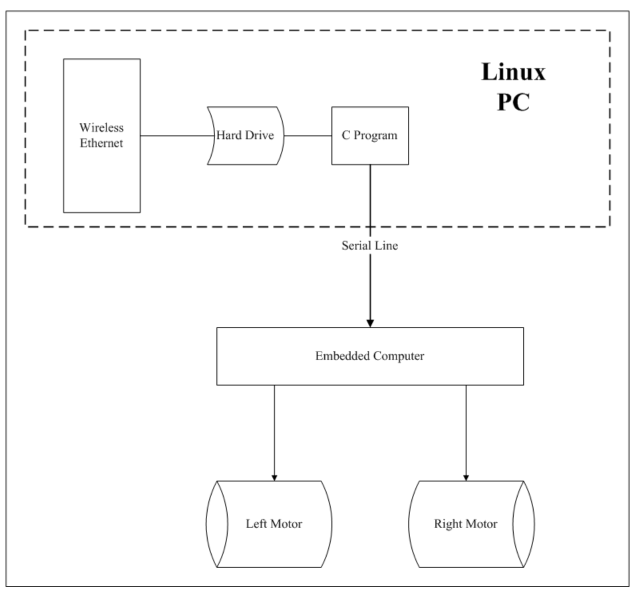


Figure 4.4: The robot hardware.

A timing file was used in order to synchronize the communication between the robot and the base station. When the base station was finished calculating the wheel velocities and writing them to a file, the base station also wrote a timing file on the robot's hard drive. The timing file contained no data, but was used as a flag to tell the small C program running on the Linux PC that new wheel velocities had been written. When the

C program was finished reading and writing the values to the serial line it would delete the timing file. Only after the timing file had been deleted would the base station write new values for the wheel velocities. The timing file kept both the Linux PC and the base station synchronized so that only one of them was trying to the read/write wheel velocities at a time. See Appendix I for the C code used to accomplish this.

4 Experimental Results

The most important result of the entire experimental setup was to prove that the motion control algorithm is efficient enough for a full scale, real-time implementation. The velocity information for a single robot was updated at approximately 7 times a second. This is well above the stated goal of greater than once per second. The update time includes the image processing time required to localize the robot, determine the velocity based on the controller, write the data and set the motor voltages. In an actual system all of these tasks or variants of them would need to be performed. The update rate of 7 Hz is an acceptable update rate for a autonomous robotic system. The proof of concept demonstration substantiates one of the goals of the project which was to create an algorithm efficient enough to be implemented in a real-time robotics system (Figure 4.5).

The implementation worked seamlessly in all aspects except at the lowest level of control. There were no bottlenecks in the vision system, the calculation of the velocities or in the communication between the robot and the base station. However, the hardware system had a difficult time setting the required motor voltages. The requested motor voltages were within specifications but the battery level could not meet the stated voltages.

The batteries consist of rechargeable Nickel Metal-Hydride (NiMH) cells with special controlling hardware. Unfortunately, the batteries have not been used extensively for two years. Unless NiMH are charged and discharged regularly, they lose their ability to store and deliver charge. This inability to hold a charge resulted in drastically reducing the amount of time the experiments could be run and also the number of robots which could be used in an experiment. In addition, the amount of motor torque available is tied to the amount of current which is delivered to the motors. As the charge level on the battery drops, the available torque also drops. With low torque, the robots had a difficult time executing a normal turn and so fresh batteries had to be swapped into the robots so that they could complete the turns required by the controller. Each swapping of a battery required that the robot be shutdown, restarted and the control programs re-initialized.

The batteries are not a standard battery which can be commercially purchased. New batteries must be purchased from the original company at exorbitant prices ($ 1,089). The batteries contain hardware which cannot be reproduced or reverse engineered. However, a solution to the battery life problem was designed by replacing the individual cells within the battery. This was not a factory intended procedure, but appears to be a cost effective way to extend the life of the batteries. See appendix K for the technical specifications of the individual cells used for replacement. The total cost of replacement was only $67.

Because of the issues of the battery life, it took approximately two hours to have one experiment run to completion. The experimental time would not allow enough experiments to be run in order to collect meaningful data given the timeline of the project. In addition, because of the motor torque issues, it would be nearly impossible to control the experiments so that they would be repeatable. The results of the experiment would be more dependant on the charge level of the battery rather than the ability of the controller or control laws.

The implementation phase of the project was completed, delivering the stated objective of creating a proof of concept demonstration of the controller on physical robots. Because of the battery life issue, pursuing this avenue of research would not be prudent. Instead a more in depth focus on simulation was explored since this would produce the most fruitful research.

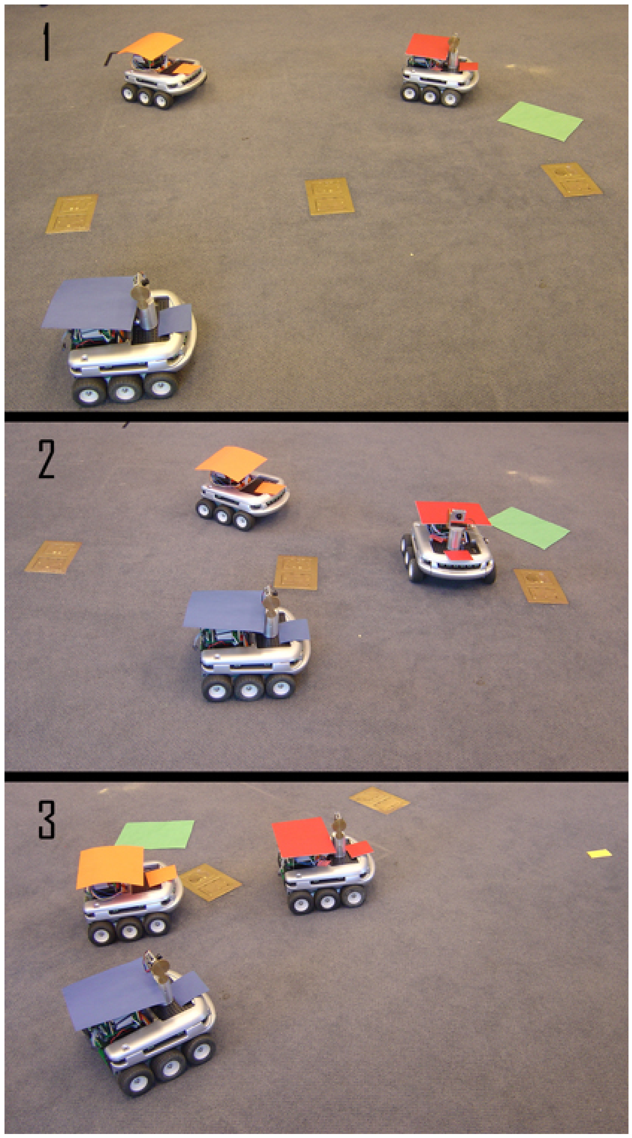


Figure 4.5: Three still frames from an experiment showing the robots coming together for communication, avoiding obstacles (green square) and moving towards the goal location (yellow square in frame 3).

Chapter 5

Experimentation through Simulation

As a large swarm of robots move about the workspace, the control laws presented earlier cause the robots to maintain line of sight and range to other robots. When both of those conditions are met a communication link can be established. When multiple links are created, the swarm as a whole constitutes a mobile wireless communication network. The number of robots needed to illuminate trends in the properties of these networks and the number of experimental trials required to ensure the results were statistically significant precluded hardware based experimentation. In this chapter large scale simulations (10+ robots) are used to understand the emergent swarm wide behavior under the influence of the control laws presented earlier.

Since robustness is frequently cited as a major motivation to employ swarm robotics, the robustness of the resulting network was then analyzed using various connectivity metrics from mathematical graph theory. The connectivity metrics of the swarm can be used to quantify the redundancy of the communication network with respect to the failure of individual robots. Such an analysis was one of the promised deliverables of the project. In this chapter we also explore the tradeoff between redundancy and task completion as measured by the time required for each robot to reach its goal.

1 Modeling robot swarms as graphs

A mathematical *graph* $\mathcal{G}$ consist of vertexes $\mathcal{V}$ (also called nodes) and edges $\mathcal{E}$. For convenience in discussion, the graphs's n vertices are labeled $\mathcal{V} = \{1, 2, \ldots, n\}$ although the labels themselves are arbitrary. An edge is essentially an unordered pair of vertices $e = (i, j)$ with $i, j \in \mathcal{V}$ and $e \in \mathcal{E}$.

A graph with n nodes can be represented numerically by an n by n matrix called a connectivity matrix, C. If nodes i and j are connected by an edge then $C_{i,j} = 1$. $C_{i,i} = 0$ by convention. C is symmetric along the main diagonal. Using this matrix, several numerical measures of the graph can be determined.

The swarm can be modeled as a graph. Each robot is a vertex. An edge exists between two vertices (robots), if and only if both range and line of sight are established. An edge represents a communication link (Figure 5.1 and Figure 5.2 a). Note that the graph theoretic representation does not encode any specifics about the positions or velocities of the robots or obstacles. It is simply an abstract representation which models the connection *topology* of the swarm.

2 Graph theoretic measures of connectivity

A graph is said to be *connected* if and only if there exists a path from every node to every other node. If a swarm is connected a single communication network exists. If the graph is disconnected, several connected sub-networks may exist.

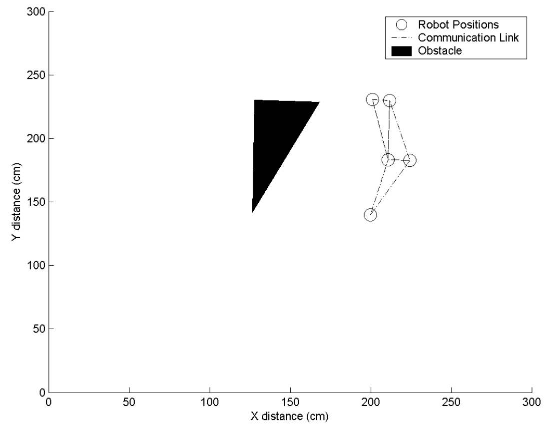


Figure 5.1: A communication network created by 5 robots in a simulation.

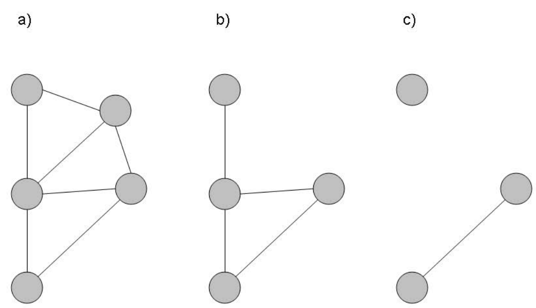


Figure 5.2: a) A graph, $\mathcal{G}$ which has 5 nodes and 7 edges. b) $K(\mathcal{G}) > 1$ because the removal of any node does not disconnect the graph. c) $K(\mathcal{G}) = 2$ because with the removal of 2 nodes the graph becomes disconnected.

The measure of connectivity which this project is most interested in is the *K–connectivity*, $K(\mathcal{G})$. The K value is the minimum number of nodes which when removed, disconnect the graph. If a graph is connected and removing one node will disconnect the graph then the graph's K value is 1 (also called a 1–connected graph). If it takes two nodes to be removed before the graph becomes disconnected then the K value is 2, etc. The K–connectivity is a worst case scenario measure, it represents the minimum number of nodes which must be removed before the graph becomes disconnected (Figure 5.2).

This property is of special interest to the project because as Criteria 4:*Fault Tolerance* states, the controller should be able to survive a massive communication disturbance such as the loss of a node (robot). The K value explicitly defines how many nodes (robots) can be lost before the swarm no long is connected. In order to for a network to truly be fault tolerant, it must be at least 2–connected, implying the swarm's connectivity hinges on no single robot.

However, four drawbacks make K insufficient as a stand-alone measure of network robustness or redundancy.

1. Most importantly, no motion controller that satisfies the project's design Criteria 3:*Distributed Operation* can directly influence K, because computing K requires knowledge of every robot's position. K is a global measure.

2. K is an overly conservative measure of connectivity. It represents a worst case scenario – the minimum number of vertex failures which disconnect the swarm. Two swarms can have identical K values, however, one of the swarms may have many more edges making it much more robust to single robot failure.

3. The connectivity of the swarm varies as a function of time due to the swarm's motion. Because K only takes on integer values it may jump from $K = 2$ to $K = 3$, for example. This is another manifestation of K's conservancy. Without some intermediate measure of connectivity it is difficult to determine a trend line.

4. Finally, K is difficult to compute because it requires a series of depth–first graph searches.

For these reasons five additional metrics were used to determine the overall connectedness of the swarm. They are:

- minimum degree,
- algebraic connectivity,
- average degree,
- number of groups of robots when the graph is disconnected,
- percentage of time in which every robot has at least two connections (simple redundancy).

Some of these measures require further explanation. The minimum degree, average degree, and algebraic connectivity are explained below.

The *Degree* of a vertex $\delta(i)$ is the number of edges incident on that vertex. In the case of robot swarms, this is the number of communication links a given robot i has established at any given time. This can be computed from the matrix C by

$$\delta_i(\mathcal{G}) = \sum_{j=1}^{n} C_{i,j}.$$

The desired degree of a given vertex is important because the minimum acceptable value for the number of communication links for a certain robot can be easily specified as an input to the parallel composition controller.

The *Minimum Degree*

$$\delta_{\min}(\mathcal{G}) = \min_{i\in[1,\ldots,n]} \sum_{j=1}^{n} C_{i,j}$$

is very important because by Whitney's inequality ([17, page 43]) and theorem 6.8 of [4]:

$$\max(0, 2\delta_{\min}(\mathcal{G}) + 2 - n) \leq K(\mathcal{G}) \leq \delta_{\min}(\mathcal{G})$$

($\max(0, 2\delta_{\min}(\mathcal{G}) + 2 - n)$ can be abbreviated as $\underline{\mathrm{K}}$). Thus, the minimum degree provides upper and lower bounds on the K–connectivity. Note however that these bounds are highly conservative. It is also of interest because by assigning each robot a desired minimum acceptable degree, one can directly control the swarm's K–connectivity in a distributed fashion.

Another property that can be extracted from C is the *Algebraic Connectivity* [12], $e_2(\mathcal{G})$. It is equal to the second smallest eigenvalue of the Laplacian matrix of C. There are several features of this property which make it attractive to use. First, $e_2(\mathcal{G}) = 0$ if and only if the graph is disconnected. According to [15]

$$e_2(\mathcal{G}) \leq K(\mathcal{G}).$$

Therefore, this sets a lower bound on the value of K. K is a worst case scenario measure and only moves in integer steps. More edges can be added to a graph, but the K value will not necessarily increase. However, the value of e_2 can be any real number and follows the number of edges. Thus, e_2 provides a less conservative measure of the overall interconnectedness of a graph. It is also very convenient to compute. It can be shown that the algebraic connectivity is related to the *Average Degree*

$$\delta_{avg}(\mathcal{G}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{i=1}^{n} C_{i,j}.$$

Thus in summary, while the K value is the primary interest in this project, it cannot be set directly and is an overly conservative comparative measure of robustness. Other measures of connectivity are more telling. The two most important are the minimum degree and the algebraic connectivity. The minimum degree can be set at a desired value by changing the desired number of communication links per robot in the parallel composition algorithm subroutine. The minimum degree indirectly effects K by setting upper and lower bounds on its value. The algebraic connectivity is continuous and proportional to the number of edges in the graph. It is highly convenient to measure and provides a tighter lower bound on K. These measures are related

$$\max(0, 2\delta_{\min}(\mathcal{G}) + 2 - n) \leq e_2(\mathcal{G}) \leq K(\mathcal{G}) \leq \delta_{\min}(\mathcal{G}).$$

Furthermore since K is always an integer and e_2 is a real number, e_2 can be rounded up to give an even tighter bound.

As an example, the graph in Figure 5.2 a has values of $\underline{\mathrm{K}} = 1$, $e_2 = 1.5858$, $K = 2$ and $\delta_{\min} = 2$.

3 Experimental procedure

A series of simulations were run to determine how changing the number of desired links per robot affects the K value. In the experiments, the desired number of links per robot (M) was the independent variable and the six metrics listed above were the dependant variables. Other variables were controlled to provide a meaningful result. All simulations used the same set of 30 different workspaces. Each workspace was a 500 by 500 centimeter box with two obstacles and 15 robots. The robots were placed randomly within a 200 by 200 *cm* box in the lower left hand corner. Each robot was assigned a unique goal position. These goal positions were randomly placed in a 100 by 100 centimeter box in the upper right hand corner. One obstacle was a quadrilateral while the other was a triangle. The obstacles were placed by hand to ensure variety in the workspace composition. In some workspaces the obstacles were placed in the direct line of the goal, while in other a narrow passage existed in direct line to the goal and in some, the obstacles were placed so they wouldn't interfere too much with the robots moving towards the goal. Figure 5.3 shows some sample workspaces while Figure 5.4 is simulation in progress.

A desired minimum degree was set at the start of a experiment. Thirty simulations, one for each workspace, would be run for 1000 iterations each. At each iteration the metrics were checked to determine robustness of the swarm communication network. At the end of an experiment, six 30 by 1000 matrices of data points were created. The data was then averaged over the 30 runs. Experiments were conducted on both controllers with zero through eight desired links per robot. See Appendix N for MATLAB computer

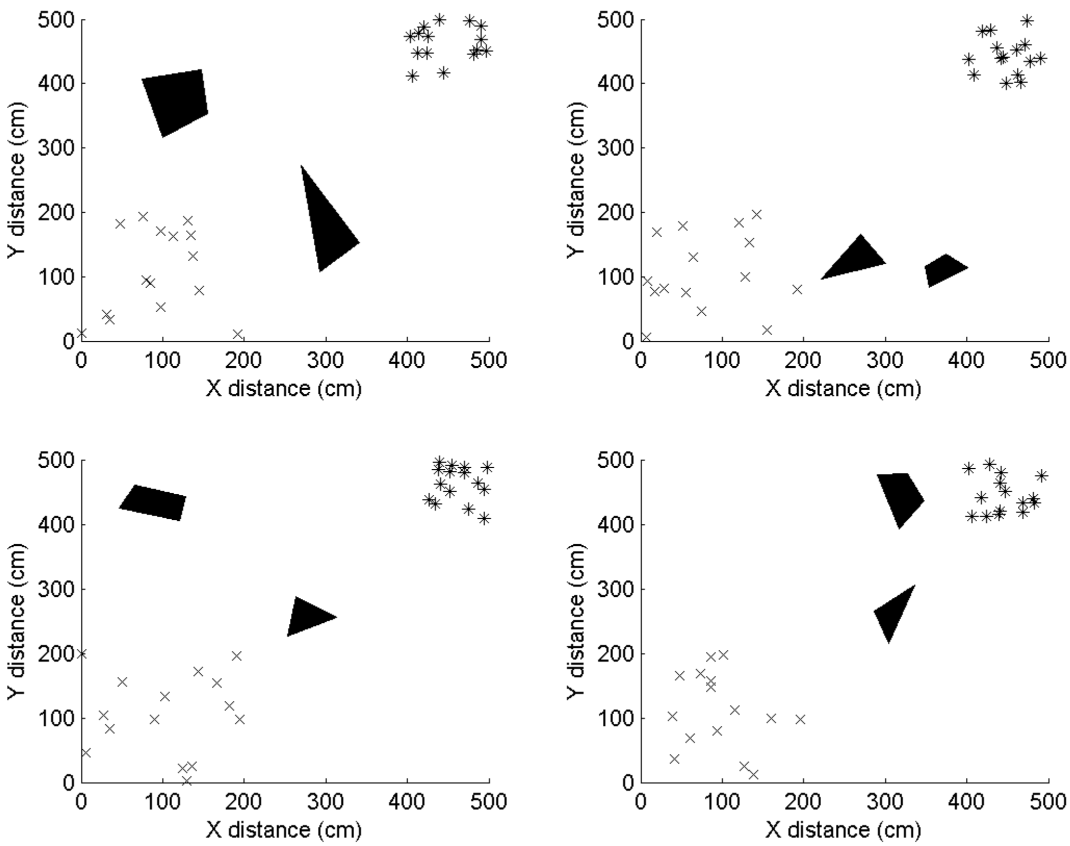


Figure 5.3: Four different workspaces. The x's mark the starting positions of the robots while the * are the goal positions.

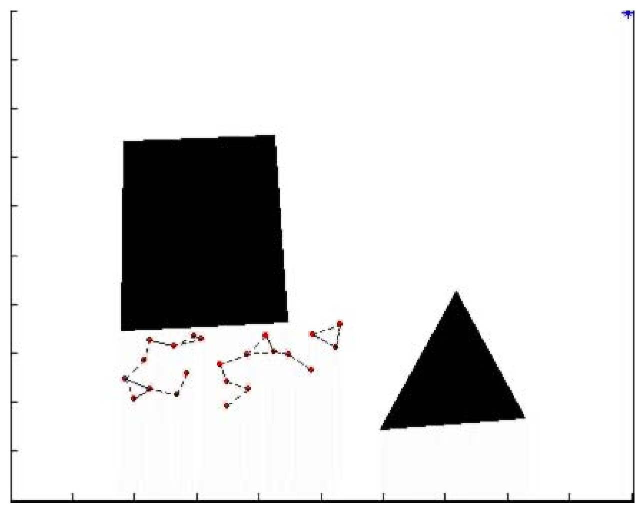


Figure 5.4: A simulation in progress with the robots forming a communication network while avoiding obstacles.

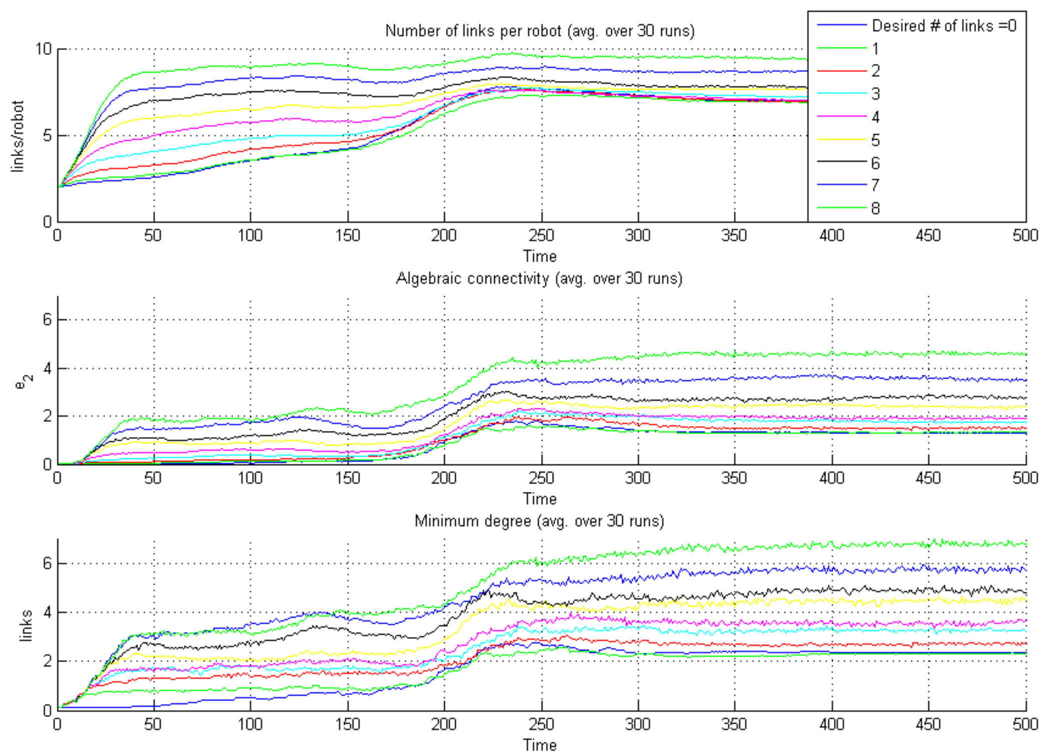


Figure 5.5: The metrics plotted over time for each experiment with the communication variant.

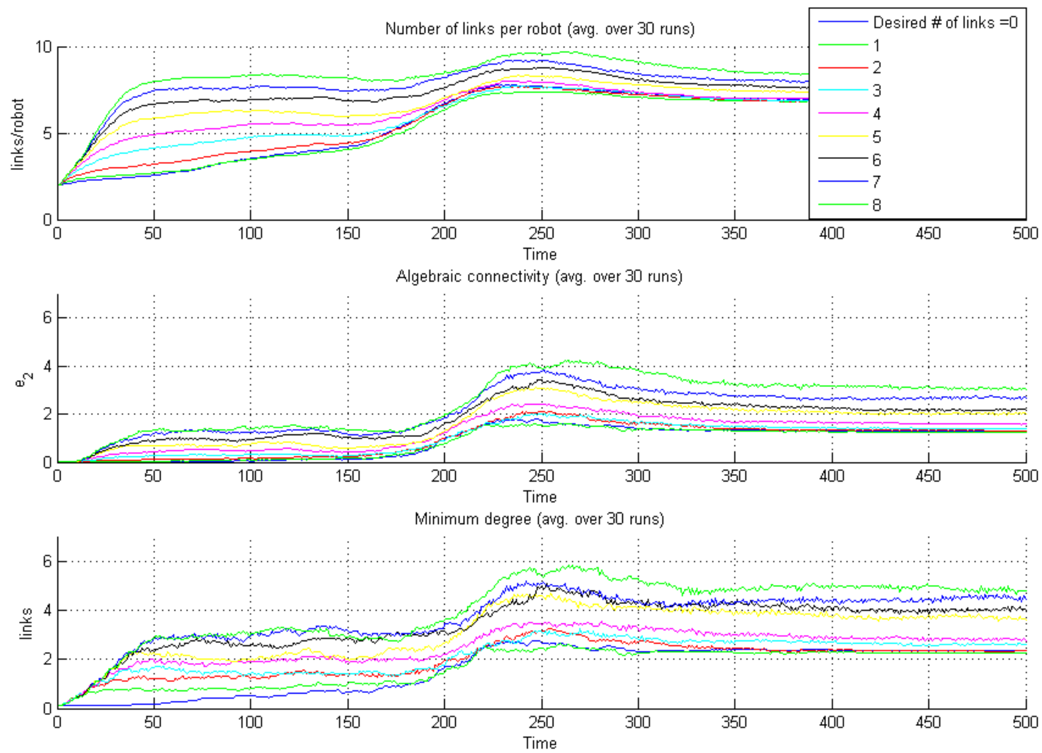


Figure 5.6: The metrics plotted over time for each experiment with the original controller.

code used to determine the metrics. Each of the averaged values for each experiment is plotted over time in Figures 5.5 and 5.6.

Eight desired communication links per robot was set as the maximum value tested. If the actual number of links equals the desired number of eight, then $K \geq 2$ for a swarm of 15 robots based on the value of $\underline{K}$. This project is most interested in creating a minimum K value of 2 because it means that no single robot failure will disconnect the swarm. At $M = 8$ the actual number of communication links was much higher than 8 because the robots were in such close proximity, therefore, increasing the number of communication links beyond 8 would not produce anything more meaningful.

The experiment with $M = 0$ simulates a swarm in which each robot moves independently, with no regard for swarm wide networking. This was used as the baseline to determine how much of a difference the controllers made in establishing a communication network. As the robots move around, and especially given the workspace set up, some communication links will be established by happenstance. The baseline results provide a measure of how robust a natural communication network will be when each robot is independently moving towards the same general area. The workspaces could have been carefully constructed so that no communication links would form by happenstance. However, this is not necessary, all that is necessary is that a baseline is established and then to determine the amount and degree to which the controller deviates from that baseline.

4 Experimental results

It was predicted that the more connections desired per robot, the greater the robustness of the swarm but also the longer it would take the swarm to reach its goal. In addition, it was believed that the robustness of the swarm would hit a saturation level and increasing the desired number of links would not effect the robustness after that saturation level. Additionally, it was predicted that the communication variant would establish a stronger network at the price of increased time to goal. While these were the qualitative predictions, no educated quantitative prediction could be made.

When examining the data, some of the metrics do not provide much useful information. The measure of simple redundancy increased until it reached saturation at $M = 2$. The number of groups also did not provide much useful information, for all of the experiments where $M \neq 0$ the number of groups quickly converged to 1.

For the meaningful metrics, the data was averaged over the first 400 iterations. The first 400 iterations were chosen because by this point, in all experiments, the robots had settled to steady state values. This project is more interested in the transient response of the controller, i.e. the response of the controller as the robots move around. The averages for this data are shown in Figure 5.7. In addition, the data for the average time to goal is also plotted. A better interpretation of the data is based on the percentage increase

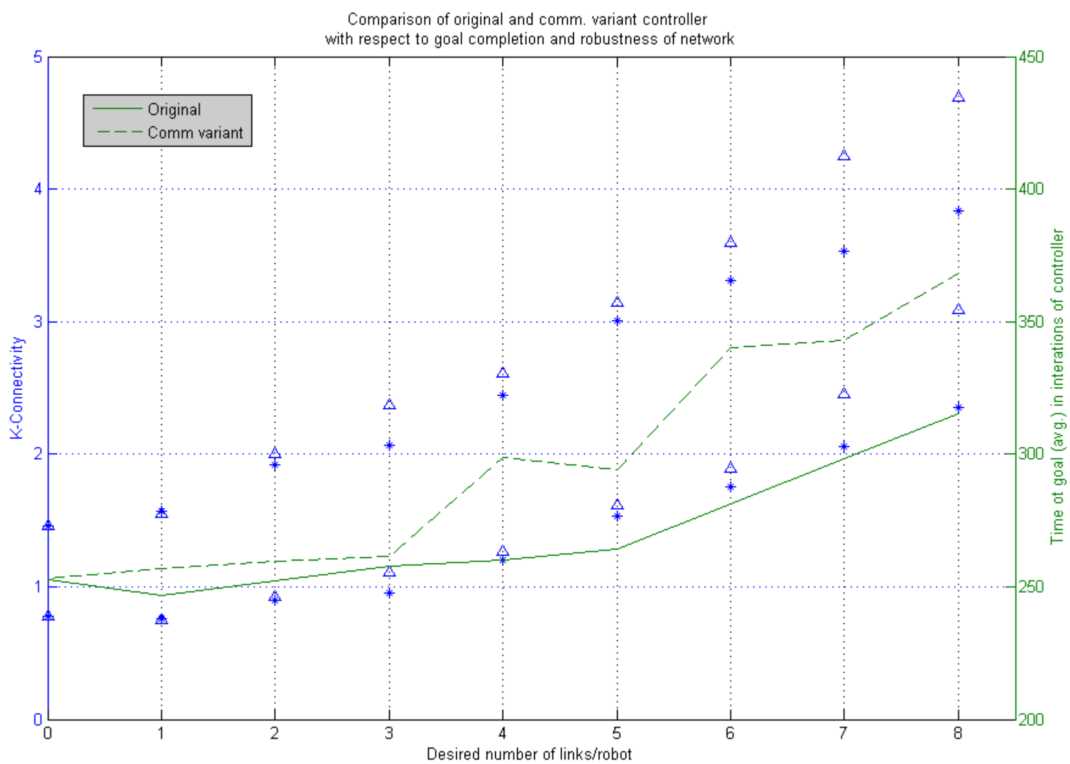


Figure 5.7: The values for the connectivity and time to goal metrics for every experiment. Triangles mark upper and lower bounds for K–connectivity of the Comm. variant, while * mark the the upper and lower bounds of the original controller's K-connectivity. Note the different y axes for time to goal and K–connectivity.

in the various values. The baseline results are simply a result of the workspace design; therefore, it is more meaningful to examine the percentage increase rather than the absolute values of the data. Therefore, Figure 5.8 is a comparison of the percentage increase in robustness and time to goal for all the experiments.

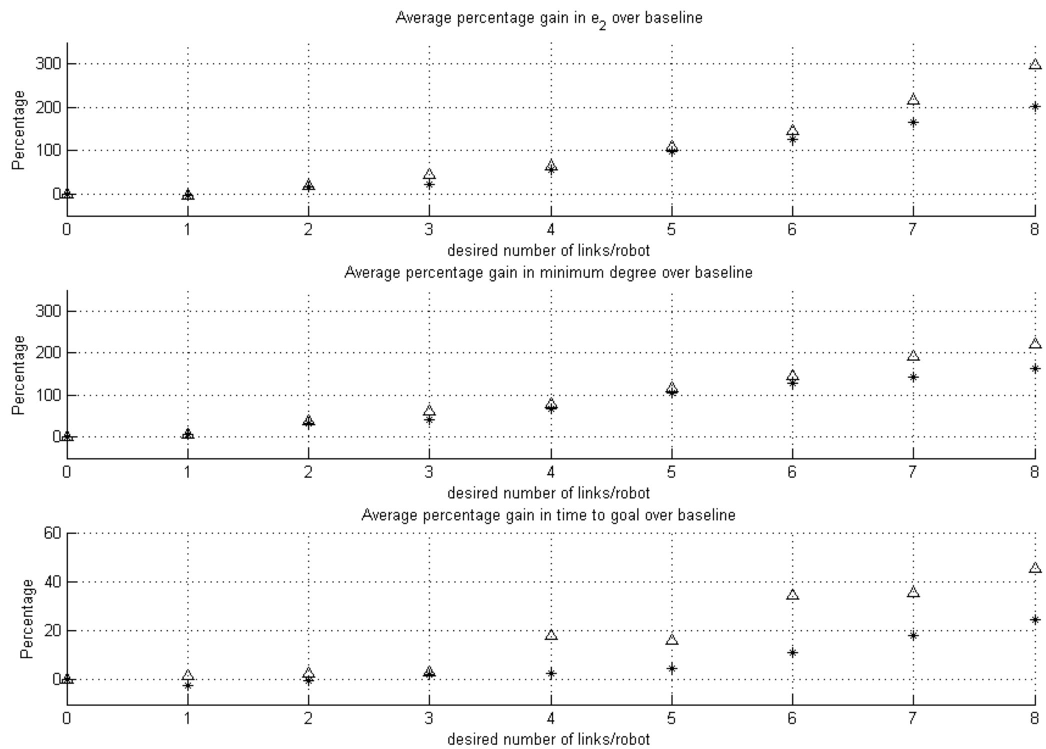


Figure 5.8: The percentage difference in the metrics between the baseline and the various experiments.

Only demanding each robot to have one communication link did little to affect the robustness of the swarm or the time to goal. It appears that the controllers reach saturation for values of $M > n/2$ but without experimental data, this cannot be proven conclusively. Also as predicted, the communication variant consistently established a more robust network than the original or goal controller. The communication variant had a larger impact when the number desired number of communication links was larger. However, this increased robustness cost the swarm an approximate 20% increase in the time to reach the goal.

It is also interesting to note that while the conservative lower bound is at zero for most of the experiments, the e_2 metric is consistently higher than the conservative lower bound. This means that the controller is consistently pushing the connectivity up towards a higher value. In some cases, $\delta > M$. This happens because as the robots pull themselves into a communication network, other communication links are established by the fact that the robots are now in closer proximity.

When taken as a whole, the controllers developed in this project, can increase the robustness of a communication network at least 200- 300% while only costing a 25-45% increase in the time to accomplish the goal. Depending on if the mission is time critical or if it needs good communication, the controller could be tuned to meet the demands.

4.1 Sources of error

The data is averaged in each of the experiments. Thus, few guarantees can be made about the performance of an individual swarm, only average statements can be made. Given another workspace, the swarm could perform below its average for its given controller. This controller is dependent on the initial conditions of the workspace and how that affects communication links being created by happenstance. By analyzing the percentage increase over the baseline, the effect of the initial conditions should be eliminated.

$\underline{\mathrm{K}}$ reflects the relationship between M and a lower bound. However, this relation is very conservative and does not provide an accurate reflection of the K values in a simulation. A more accurate formula which relates Q to a lower bound of K is desired. The network created is a special type of graph called a proximity graph so perhaps by using the properties of this type of graph, a tighter theoretical constraint on K could be established. Time limits prohibit this investigation.

Chapter 6

Conclusion and Future Work

1 Conclusion

This project designed a solution to the stated task of creating an overall motion control algorithm which can accomplish the dual, and sometimes conflicting requirements of executing a primary mission while at the same time establishing and maintaining a robust communication network. This project created several potential fields to accomplish this, each of which is an original theoretical contribution. Additionally, a novel high level control algorithm was created and implemented. The parallel composition scheme took a theoretical (nonconstructive) proof and exploited the planar structure of the problem to develop a computationally efficient: 1. feasibility test and 2. method of computing velocity vectors.

Additionally, two different algorithms were developed which prioritize and determine a feasible set of vectors. Both allow the most urgent and important vectors to be reflected in the final velocity. One is optimized for communication and the other for goal completion.

Several hundred (510) simulations were run to test and analyze the performance of the controller at accomplishing the goal and maintaining a robust communication network. The results of the simulations prove that the controller is able to accomplish its dual objectives simultaneously. Quantitative analysis shows that using the controller can increase the connectivity of the swarm, as compared to the baseline experiments, by 200-300% while only incurring a 25-45% increase in the time it takes to reach the goal.

The controller was also implemented on a physical robot system to test how the controller could be implemented on real robots. Even with all other component necessary for the robot system, the algorithm was fast enough that it could update a robot's speed 7 times a second. This proved that the algorithm was efficient enough to be used as a real-time controller for robots.

The functions, algorithms, and controllers created in this project accomplish the task goals in way that was designed for swarms of multiple, unsophisticated, independent robots. The functions, algorithms and controllers accomplish criterions of *Goal Completion* and *Robust Communications* in a manner that satisfied the criterions of *Distributed Operation* and *Fault Tolerance*.

In summery, the accomplishments of this project are as follows:

Theoretical accomplishments

- Designed a distributed control law for maintaining inter-robot separation between two robots. This control law allows for the minimum separation distance and maximum communication distance to be set independently.

- Designed a distributed control law for maintaining line of sight between two robots. This control law maintains line of sight and if two robots lose line of sight, it pushes one of the robots out of the occlusion zone and back into line of sight communication.

- An algorithm was designed which enforces the redundancy constraint and maximizes the number of task objectives which can be meet while minimizing the energy used.
- An original, unique and novel high level control scheme was developed which facilitates meeting two or more objectives simultaneously and reflects the importance of meeting the objectives.

Implementation and design accomplishments

- Designed a computer vision system capable of identifying and orienting the robots in $< .1$ seconds.
- Engineered a framework for controlling the Koala Robots, including a unique control methodology using MATLAB, wireless Ethernet, and the C programming language.
- Demonstrated a physical system of robots to prove that the control algorithm could be implemented and used on a real time robotics system.

Experimental analysis accomplishments

- Performed hundreds of simulations in various workspaces using large numbers of robots to test the performance of the controller.
- Pursued in depth quantitativeness analysis using graph theory techniques and formulas to determine the robustness of the swarm communication network.
- Demonstrated through simulations that the control algorithms can cause a 200-300% increase in the robustness of the swarm while only incurring a 25-45% increase in the time to complete the goal.

2 Ideas for future research

There are still many avenues to extend the results of this research. The experimental K value is much higher than the theoretical lower bound of $\underline{\mathrm{K}}$. Determining a less conservative relationship between K and δ is a top priority. This would lead to a method of selecting δ, to achieve a certain degree of redundancy. The robot's communication network is a special type of graph called a proximity graph. Perhaps by examining the properties of these graphs a tighter relationship could be found.

One aspect that this project explored, but due to the time limitations, only tentative conclusions could be drawn, is the idea of only having restricted position information. With restricted position information, a robot only knows the position information of the robots with which it has a communication path. It is important to note that the motion control algorithm still functions without omniscience position knowledge. This scenario is equivalent to the 'eye in the sky' being removed and each robot has to communicate its position to every other robot with which a communication path exists. Not enough simulations were run to provide a definitive result. However, in initial simulations each group of robots forms the most robust communication network given the number of robots and the Q value. If by happenstance two groups should meet, then the network will rearrange itself based on the new position information which can be obtained. It is predicted that, at the beginning of a simulation, the metrics appear to be similar to the baseline controller. At the start, there are many groups and so there is little position information which constrain the robots. But as the simulation progresses, the groups become larger and thus the metrics will begin to mirror the simulations where the robots had omniscient position knowledge.

A promising future research project would be to combine this project with some of the work outlined by Poduri and Sukhatme [30]. This work examines using proximity graphs for maximizing the coverage of mobile sensing networks while keeping some degree of communication robustness. However, the proximity graphs examined are a highly ordered lattice of points which might be unsuitable for a real environment with obstacles. If the lattice points were set as the goal positions of the robots in an unknown workspace, then the controller designed in this project could be used to guide the robots towards the goal positions while maintaining a communication/sensor network. If there were any obstacles in the way of the robots reaching the lattice points, this project's controller would approximate the ideal lattice positions in such a way that the communication/sensing network would not be effected greatly. Thus, using the results on proximity graphs for coverage contained in [30], and this project's motion control algorithm, a controller for maximizing sensor converge while maintaining a robust communication network can be created.

To better approximate the real world, noise could be added into the communications. This would mean that a communication link, even though the robots are within range and line of sight, could not be established for a brief period of time. The problem of dropped communication links on the robustness of a controller is closely related to its K value. $\mathcal{E}(\mathcal{G})$ is like the K value, except it is the minimum number of *edges* that need to be dropped before the graph becomes disconnected. The relationship between K and $\mathcal{E}$ is as follows [17, page 43]:

$$K(\mathcal{G}) \leq \mathcal{E}(\mathcal{G}) \leq \delta(\mathcal{G}).$$

A final area of future exploration is asynchronous communications, where the robots do not have instantaneous updated information of the positions of the swarm members. Instead, it would take a certain amount of time for a robot to receive updated information from another robot on the opposite side of the swarm. This problem is closely related to errors in position information which come from localization errors. Incorporating these delays and errors would help increase the realism of the simulations.

Bibliography

[1] S. Anderson, R. Simmons, and D. Goldberg. Maintaining line of sight communications networks between planetary rovers. In *Proceedings of the Conference on Intelligent Robots and Systems*, October 2003.

[2] R. Arkin. *Behavior Based Robotics.* The MIT Press, Cambridge, MA, 1998.

[3] R. Arkin and J. Diaz. Line–of–sight constrained exploration for reactive multiagent robotic teams. In *7^{th} International Workshop on Advanced Motion Control*, 2002.

[4] V. Balakrishnan. *Schaum's Outline of Theory and Problems of Graph Theory.* Schaum's outline Series (McGraw–Hill), 1997.

[5] M. Barr. *Programming Embedded Systems in C and C++*, chapter 3, page 22. O'Reilly & Associates, Inc., Sebastopol, CA, first edition, January 1999.

[6] B. Bishop. On the use of redundant manipulator techniques for control of platoons of cooperating robotic vehicles. In *IEEE Transactions on Systems, Man and Cybernetics*, pages 608–15, September 2003.

[7] R. Brooks. *How to Build Complete Creatures Rather than Isolated Cognitive Simulators*, pages 225–39. Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.

[8] H.S.M. Coxeter and S. Greitzer. *Geometry Revisited*, pages 82–83. Mathematical Association of America, 1967.

[9] L. Deligiannidis and V. Tsaoussidis. Experiences in architecting redundant and secure networks. *International Conference on Internet Computing*, 2:905-14, June 2003.

[10] J. Desai, J. Ostrowski, and V. Kumar. Modelling and Control of Formations of non-holonomic mobile robots. *IEEE Transactions on robotics and automation*, 17(6):905-916, 2001.

[11] J. Esposito and V. Kumar. A method for modifying closed loop motion plans to satisfy unpredictable dynamic constraints at runtime. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, pages 1691–16, May 2002.

[12] M. Fiedler. Algebraic connectivity of graphs. *Czech. Math. Journal*, 23(98):298–305, 1973.

[13] Y. Gao, K. Wu, and Li Fulu. Analysis on the redundancy of wireless sensor networks. *International Conference on Mobile Computing and Networking*, 2003.

[14] S. Ghozati. The topological design of computer networks. *Advances in Modelling and Analysis*, 22(2):19-27, 1994.

[15] C. Godsil and G. Royle. *Algebraic Graph Theory.* Springer Graduate Texts in Mathematics, 2001.

[16] R. Grabowski, P. Khosla, and H. Choset. Development and deployment of a line of sight virtual sensor for heterogeneous teams. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pages 3024–3029, New Orleans, LA, April 2004.

[17] F. Harary. *Graph Theory*. Addison–Wesley Publishing Group, 1969.

[18] A. Jadbabaie, J. Lin, and A. Morse. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Transactions on Automatic Control*, 48(6):988-1001, June 2003.

[19] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *International Journal of Robotics Research*, volume 5(1), pages 90–98, 1986.

[20] J. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[21] Q. Li and D. Rus. Sending messages to mobile users in disconnected ad-hoc wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 44–55, August 2000.

[22] Y. Liu and K. Passino. Stable social foraging swarms in a noisy environment. *IEEE Transactions on Automatic Control*, 49(1):30–44, January 2004.

[23] M. Medard, S. Finn, R. Barry, and R. Gallager. Redundant trees for preplanned recovery in arbitrary vertex-redundant of edge-redundant graphs. *IEEE/ACM Transactions on Networking*, 7(5):641-52, October 1999.

[24] Y. Meyer. Wavelets: Algorithms and applications. *Society for Industrial and Applied Mathematics*, pages 13–31, 101–105, 1993.

[25] Office of Naval Research. Autonomous Operations of Future Naval Capabilities . http://www.onr.navy.mil/fncs/auto_ops, January 2003.

[26] R. Olfati-Saber. Flocking for multi-agent dynamic systems: algorithms and theory. *IEE Transactions on Automatic Control*, 49(2), June 2004.

[27] R. Olfati-Saber and R. Murray. Distributed cooperative control of multiple vehicle formations using structural potential functions. In *15th IFAC Wold Congress, Barcelona, Spain*, 2002.

[28] R. Olfati-Saber and R. Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on Automatic Control*, 49(9):1520–1533, September 2004.

[29] J. Piepmeier and P. Morgan. Uncalibrated vision-based control of two wheeled mobile robots. *ASME International Mechanical Engineering Congress and Exposition*, 2:1939–1944, 2001.

[30] S. Poduri and G. Sukhatme. Constrained coverage for mobile sensor networks. *International Conference on Robotics and Automation*, 1:165–171, 2004.

[31] J. Reif and H. Wang. Social potential fields: A distributed behavioral control for autonomous robots. *Robotics and Autonomous Systems*, 27:171–194, 1998.

[32] C. Reynolds. Flocks, herds and schools: a distributed behavior model. In *ACM SIGRAPH Proceedings*, volume 21, pages 24–34, July 1987.

[33] D. Rus. Distributed algorithms for navigation with sensor networks. *MOBICOM*, 2003.

[34] SRI International. The Centibots Project. http://www.ai.sri.com/centibots, October 2004.

[35] D. Swaroop and J. Hedrick. String stability of interconnected systems. *IEEE Transactions on Automatic Control*, 41(3):349–357, March 1996.

[36] J. Sweeney, T. Brunette, Y. Yang, and R. Grupen. Coordinated teams of reactive mobile platforms. In *IEEE International Conference on Robotics and Automation*, pages 299–304, Washington D.C., May 2002.

[37] Y. Tan. Synthesis of a controller for swarming robots performing underwater mine countermeasures. Trident scholar project report, United States Naval Academy, 2004.

[38] H. Tanner, A. Jadbabaie, and G. Pappas. Flocking in fixed and switching networks. *Automatica*, July 2003.

[39] J. Terpstra. *Samba-3 by Example*. Prentice Hall PTR, 2004.

[40] F. Visard. Mini Plane with a High IQ. *Popular Science*, July 2002.

[41] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 2^{nd} edition, 2001.

[42] Y. Yang, O. Brock, and R. Grupen. Exploiting redundancy to implement multi-objective behavior. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation*, pages 3385–90, September 2003.

Appendices

A Programs for obstacle avoidance

```
function [dx, dy]= Bishopobstacle(xR,yR, n, stats);
%obstacpreprocessor must have been run to define stats and idx
%This fucntion returns the velocity to avoid an obstacle based upon a
% robot's
%position.
%This code is partly based off of code written by PRof. Bishop and Yong
% Tan in their
%Trident project.
%20JAN

G = 50;  %coeffient factor of potential field.
dc = 15; %cutoff distance
%maxh = 5; %max velocity that can be returned.

dx = 0;
dy = 0;
%global stats
%global SOI

%distocentroid =
% distance(xR,yR,stats(n).Centroid(1),stats(n).Centroid(2));

%if(distocentroid <=SOI(n))%if the distance to the centroid is less than
% the Sphere of Influence

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%find the approprate two vertices

roboangle = atan2(yR-stats(n).Centroid(2),xR-stats(n).Centroid(1));

if (roboangle > max(stats(n).angles) | roboangle < min(stats(n).angles))
    i1 = 1;
    i2 = length(stats(n).angles);
else
    for q = 2:length(stats(n).angles)
        if roboangle < stats(n).angles(q)
            i1 = q;
            i2 =q-1;
            break
        end
    end
end


x1 = stats(n).vx(i1);
x2 = stats(n).vx(i2);
```

```
y1 = stats(n).vy(i1);
y2 = stats(n).vy(i2);

%plot(x1, y1, '*')
%plot(x2, y2, ' g *')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%finding distances
if (stats(n).vx(i1) ~= stats(n).vx(i2))
    slopeL = (y2-y1)/(x2 -x1);   %L is short for line of obstacle.
else
    slopeL = Inf
end

interceptL = -slopeL *x1 +y1;

A = slopeL;
B = -1;
C = interceptL;

d = abs((A*xR + B*yR +C)/sqrt(A*A +B*B)); % based on point-line distnace
% formula

angledge =atan2(y2 -y1,x2-x1);
angleRepulse = angledge +pi/2; %this works because of the way the angledge
% is defined using i1 and i2

xL = xR +cos(angleRepulse +pi)*d; %these are the points on the edge
yL = yR +sin(angleRepulse +pi)*d;

d1 = sqrt((xL -x1)^2+ (yL -y1)^2);
d2 = sqrt((xL -x2)^2+ (yL -y2)^2);
L =  sqrt((x1 -x2)^2+ (y1 -y2)^2);

%plot(xL, yL, 'm *')

if (max([d1,d2]) > L) %if the point is off obstacele
    dv1 = sqrt((xR-x1)^2+ (yR -y1)^2);
    dv2 = sqrt((xR-x2)^2+ (yR -y2)^2);
    [d indx] = min([dv1, dv2]); %redefine d
    if indx == 1;
angleRepulse = atan2(yR-y1,xR-x1); %and redefine angleRepulse
    else
        angleRepulse = atan2(yR-y2,xR-x2);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%finding velocity

if d>dc
    zprime =0;
else
    zprime = G*(1/d -1/dc)*1/d^2; %Latombe pg 300  zprime is the velocity
% field, since this one is differntable,can take a short cut
end

% if zprime> maxh
%     zprime =maxh;  %capping the output
% end
```

```
xunit = cos(angleRepulse);   %must be written this way to ensure vector is
% pointing the correct way.
yunit = sin(angleRepulse);

dx = zprime*xunit;
dy = zprime*yunit;
%end
```

B Programs for inter-robot separation distance

```
function [dz] = Impsemi(dist)  %code written for speed of runtime not ease
% of readablity

maxcom = 9;   %maximum comm range really 80cm

%if min sep needs to
%change, then c,xsig and h need to be changed accordingly.

if (dist > maxcom)    %if x is outside of max comm range, use parabola
    %parabola
    K=.2;   %parabolic coeffient
    dz = 2*K*(dist-maxcom);
    return
end

dz =0;
```

C Programs for line of sight

```
function [dx,dy] = LOSvelf(x,y, gammab, gammat, outside)
%
% figure(2);clf; hold on;
%   quiver(0, 0, 50*cos(gammat), 50*sin(gammat), 'r')
%   quiver(0, 0, 50*cos(gammab), 50*sin(gammab), 'b')
%   plot(x,y, 'm *')

cutoff =.5;  %how low the speed should be at the occulsion lines
scale = 1;   %just a factor to scale the speed by, speed * 1/scale

% theta = (gammat+gammab)/2;
%
% roboangle = atan2(y,x);

intercept = 0; %this is only because of the way I've set it up, in real
% implimentation it needs to be calced from other bots position
    %However, if the other robot is set at 0,0 and x,y are
% adjusted
    %accodingly, this will always be zero

A = tan(gammab);
B = -1;
C = intercept;

db = abs(A*x + B*y +C)/sqrt(A*A +B*B)*1/scale; % based on point-line
% distnace formula

A = tan(gammat);
%B = -1;
%C = intercept;

dt = abs(A*x + B*y +C)/sqrt(A*A +B*B)*1/scale; % based on point-line
% distnace formula

d =min([db dt]);   %find which occlusion line is closer

if db <= dt;
    gama =gammab;
    slopea = gama -pi/2;
else
    gama =gammat;
    slopea = gama +pi/2;
end

%outside

if (outside);
    d = -d;
end

d = d +cutoff;

if d< 0
    d =0;
end

% d;
% slopea;
% gama;
```

```
dx = cos(slopea)*(d);
dy = sin(slopea)*(d); %this means speed is proportional to distance from
% occlusion line

% quiver(0, 0, 50*cos(slopea), 50*sin(slopea), 'k')
%quiver(x,y,dx,dy);
%axis equal
```

```
function [outside, sanctum, gammab, gammat] = LOSconnect(x,y,A);
% this function takes the verticies of the convex hull and coverts them
% into vectors, it then finds the two occulsion vectors in the same way
% that the smart sum does, by looking for a jump greater than 180. because
% these are convex obstacles, the occlusion zone will always be less than
% 180 degrees.
%
%---------------------------------------
%everything for this code is in 0 - 2*pi radians

% determine number of vertices
S = size(A);
N = S(2);

theta(:,1) = atan2(A(2,:), A(1,:))';

i = find(theta<0);
theta(i) = 2*pi+theta(i);

% use column vector ...it should be one already?
%theta = theta'

% now sort by increaseing theta, ind lets you reference back to actual
% vectors in A later
[theta ind] = sort(theta);
%what's going on here? %everythings okay, sort works on column vectors
%theta = theta';

% looking at change in theta as we go through list
for q =1:(N-1)
    deltaTheta(q) = theta(q+1) - theta(q);
end
deltaTheta(N) = (2*pi- theta(N)) + theta(1);

maxDtheta = max(deltaTheta);

outside =true; %assumed outside until proven otherwise
if (maxDtheta == deltaTheta(N))  %if N is the largest angle, then this
% means that the first is the smallest since angles are in increasing
% order
    gammat = theta(N);  %no wrap going on here,
    gammab = theta(1);
    v1x = A(1,ind(N));
    v1y = A(2,ind(N));   %find the constraining vertice points
    v2x = A(1,ind(1));
    v2y = A(2,ind(1));
```

```
    roboangle = atan2(y,x);
    if roboangle <0;
        roboangle = roboangle+2*pi;
    end
    if ((gammab) < roboangle && roboangle <(gammat))  %this is an and
outside =false;  %then it is inside occlusion lines
    end

else
    j = find(deltaTheta == maxDtheta);
    %'wrap'
    gammat = theta(j); %This is the angle which is the smallest one
    gammab = theta(j+1);  %this is the largest angle, however, it will be
% on the 'bottom' because of wrap
    v1x = A(1,ind(j));  %hence why it is called the gammab
    v1y = A(2,ind(j));   %following the above convention of gammat is v1
    v2x = A(1,ind(j+1));
    v2y = A(2,ind(j+1));

    roboangle = atan2(y,x);

    if roboangle <0;
        roboangle = roboangle+2*pi;
    end

    if (roboangle <gammat | gammab < roboangle)  %for all other cases, it
% is an and
        outside =false;
    end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%now that we have an upper and lower radius, and upper and lower occlusion

% figure(2)
% %clf
% hold on;
% plot(A(1,:)',A(2,:)', 'k');
% plot(x,y,'^');


%%%%%%%%%%%%Note that we have converted to radians from here on out

sanctum =true;
if (~outside)  %if it is inside the occulsion lines
    %is it in the inner sanctium?
    slope = (v2y-v1y)/(v2x -v1x);
    xinter = (v1y - slope*v1x)/(tan(roboangle)-slope);
    yinter = tan(roboangle)*xinter;

    %quiver(0,0,xinter,yinter)
    %quiver(0,0,x,y)
    interdist =sqrt(xinter^2 +yinter^2);
    robodist = sqrt(x^2+y^2);

    if( robodist > interdist)  %if it is outside of inner sanctum
        sanctum =false;
% it is not in the inner sanctum
    end
end
```

```
%in LOS communication = sanctum;  %it is only out of LOS comms if it is
% inside the occulsion lines & not in the inner sanctum!
```

D Programs for redundancy

```
function[dzdx, dzdy] = circumcenter(x,y,xR,yR);

%find the nearest two bots
for i =[1:length(xR)]
    distances(i) = distance(x,y,xR(i),yR(i));
end

[d i] = min(distances);
dist1 = d;
i1 = i;      %finding shortest distance indicex

distances(i) =1000000;

[d i] = min(distances);       %finding 2nd shortest distance
dist2 = d;
i2 =i;

xc1 = (x+xR(i1))/2;
yc1 = (y+yR(i1))/2;
xc2 = (x+xR(i2))/2; %finding the bisector point of the lines
yc2 = (y+yR(i2))/2;

slope1 = -(xR(i1)-x)/(yR(i1)-y);    %constructing a perpendicular line
slope2 = -(xR(i2)-x)/(yR(i2)-y);

b1 = slope1*-xc1 + yc1;
b2 = slope2*-xc2 + yc2;

xcircum = (b1-b2)/(slope2-slope1);
ycircum = slope1*xcircum + b1;

%figure(1)
axis equal
hold on;
plot(xcircum, ycircum, '. g')

d = distance(x,y,xcircum,ycircum);
% distance(xR(i1),yR(i1), xcircum, ycircum) %all these should be the same
% as d
% distance(xR(i2),yR(i2), xcircum, ycircum)

K = 1;  %Spring constant

%z = .5* K * d^2;   %to use regular simulation program, based on
% gradients, comment everything after this
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

xunit = (xcircum - x)/d;   %this is a quicker way to calculate the
% gradient, b/c I know where it is ending up.
yunit =(ycircum -y)/d;

%zprime now becomes a velocity field, take dirivative of z

zprime = K*d;  %zprime is the velocity

dzdx = zprime*xunit; %velocity times unit vector length
dzdy = zprime*yunit;
```

```
function [dzdx,dzdy,xfc,yfc] = fermatcenvel(x,y,xR,yR);
%this fucntion calculates the fermat center and also cacluates the
% velocity
figure(2); hold on;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%find the nearest two bots

for i =[1:length(xR)]
    distances(i) = distance(x,y,xR(i),yR(i));
end

[d i] = min(distances);
dist1 = d;
i1 = i;      %finding shortest distance indicex

distances(i) =1000000;

[d i] = min(distances);       %finding 2nd shortest distance
dist2 = d;
i2 =i;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%finding angles of triangle made by the 3 bots
%vetrexies are x = (x,y)
%1 =(xR(i1),yR(i1) and
%2 =(xR(i2),yR(i2)

ax =...
acos(dot([xR(i1)-x,yR(i1)-y],[xR(i2)-x,yR(i2)-y])/(norm([xR(i1)-x,yR(i1)-y...
])*norm([xR(i2)-x,yR(i2)-y])));

a1 =...
acos(dot([x-xR(i1),y-yR(i1)],[xR(i2)-xR(i1),yR(i2)-yR(i1)])/(norm([x-xR(i1...
),y-yR(i1)])*norm([xR(i2)-xR(i1),yR(i2)-yR(i1)])));

a2 =...
acos(dot([xR(i1)-xR(i2),yR(i1)-yR(i2)],[x-xR(i2),y-yR(i2)])/(norm([x-xR(i2...
),y-yR(i2)])*norm([xR(i1)-xR(i2),yR(i1)-yR(i2)])));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%if one of the angles is larger than 120, fermat point is at that angle's
%vertex

[m index] = max([ax,a1,a2]);

if (m >= (120*(pi/180)))    %if it is very obtuse then set xfc,yfc at the
% vertx point

    if index ==1
        xfc = x;
        yfc = y;
    end

    if index ==2
        xfc = xR(i1);
        yfc = yR(i1);
```

```
    end

    if index ==3
        xfc = xR(i2);
        yfc = yR(i2);
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%otherwise calculate the fermat center

else


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%constructing points on equalateral triangles and constructing lines.

    angle1 =atan2(yR(i1)-y,xR(i1)-x);
    angle2 =atan2(yR(i2)-y,xR(i2)-x);

    if abs(angle2-angle1) > pi
        if(angle1<angle2)
    pangle = angle1+60*pi/180;     %if the difference between the
% angles is greater than 180(or 240 more pecisely)
    qangle = angle2-60*pi/180;      %than the smaller of the
% angles gets +60 added to it
        else
            pangle = angle1-60*pi/180;
            qangle = angle2+60*pi/180;
        end
    else        %else, if the difference
% between the angles is less than 180(120)
        if (angle1>angle2)
    pangle = angle1+60*pi/180; %the larger of the angles
% gets 60 added to it
            qangle = angle2-60*pi/180;
        else
            pangle = angle1-60*pi/180;
            qangle = angle2+60*pi/180;
        end
    end

    xp = dist1*cos(pangle)+x
    yp = dist1*sin(pangle)+y   %finding the points on equalateral
% triangles

    %plot(xp,yp, '^')

    xq = dist2*cos(qangle)+x
    yq = dist2*sin(qangle)+y

    %plot(xq, yq, '^')

    slope1 = (yp-yR(i2))/(xp-xR(i2));  %constructing lines
    slope2 = (yq-yR(i1))/(xq-xR(i1));

    b1 = slope1*-xp + yp;
    b2 = slope2*-xq + yq;

      g=[0:.5:11];
     plot(g,slope1*g+b1, 'y')
     plot(g,slope2*g+b2, 'm')
```

```
    xfc = (b1-b2)/(slope2-slope1);   %solving lines for intercection pt.
    yfc = slope1*xfc + b1;
end

plot(xfc, yfc, '+ r')     %plot fermat center

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%calculate the height => potential field

%d = distance(x,y,xfc,yfc);      %calculating potential function
%K = 1;       %Spring constant
%z = .5*K*d^2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%calculate gradient => calculate velocity

d = distance(x,y,xfc,yfc);     %calculating potential function
if d ==0     %in the case that on angle is >120 d
% =0 so no movement occures, return to calling function
    zprime=0;
    dzdx =0;
    dzdy =0;
    return
end

K = 1;     %Spring constant

xunit = (xfc - x)/d;   %this is a quicker way to calculate the gradient,
% b/c I know where it is ending up.
yunit =(yfc -y)/d;

zprime = K*d;  %zprime is the velocity

if zprime >10
    zprime =10;  %velocity cap
end

dzdx = zprime*xunit; %velocity times unit vector
dzdy = zprime*yunit;
```

E Programs for parallel composition controller

```
function [feasible] = feasiblity(A)
% this function takes a matrix A
% which contains a set of N 2-D vectors concatenated to form a
% 2XN matrix.  Each vector represents the desired velocity for one
% of the potential functiosn that are currently active

%  The function attempts to compute a set of two basis vectors, such
%  that any velocity vector which lies inside their convex hull would
%  simulatneously decrease all the potential functions.

%Here are two test examples
%feasible
%A= [1 1 -.1 1;1 2 -4 -1]
% infeasible
%A= [-2 1 1 -.1 1;0 1 2 -4 -1]
%-------------------------------------

% determine number of potential fuctions
S = size(A);
N = S(2);
if N ==1 || ~N;  %it is feasible if it is exactly one or exactly zero
    feasible =true;
    return
end


for(i=1:N)
    % compute the angle of each vector
    theta(i,1) = atan2(A(2,i), A(1,i))'*180/pi;
    % theta in [0,360] format
    if (theta(i)<0)
        theta(i) = 360+theta(i);
    end
end
% use column vector ...it should be one already?
theta = theta';

% now sort by increaseing theta, ind lets you reference back to actual
% vectors in A later
[theta ind] = sort(theta);
%what's going on here?
theta = theta';
theta';

% looking at change in theta as we go through list
deltaTheta = [theta(1)-theta(N); theta(2:N)- theta(1:N-1)];
%again make sure that deltaTheta in [0,360]
for(i=1:N)
    if (deltaTheta(i)<0)
        deltaTheta(i) = 360+deltaTheta(i);
    end
end

[maxDtheta, j] = max(deltaTheta);

if (maxDtheta<=180)
    %'infeasible'
    feasible = false;
else
    %'feasible'
```

```
    feasible = true;
end
```

```
function A = paracontrollerupdate(P,C, commsindx, commnum)  %P is a 2-N
% priority matrix
%which is in the format of rows: dx;dy
%C is a 4-N comm matrix which is in
%the format, rows: dxLOS;dyLOS;dxrange;dyrange

%Tom Dunbar
%Trident Project
%24MAR

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5
%Check if all priority 1 velocities are feasible


feasible = feasiblity(P);

A=P;
while(~feasible) %while it is not feasible
D=[];
veloA =[];  %blanks it out or else an infinite loop will occur!!!
    S = size(A);
    for i = 1:S(2)
        veloA(i) = norm(A(:,i));
    end

    [velo i] = min(veloA);
    D(1,:) = [A(1,1:i-1),A(1,i+1:end)];  %delete the smallest velocity
    D(2,:) = [A(2,1:i-1),A(2,i+1:end)];
    feasible = feasiblity(D); %recheck feasibility
    A =D; %after the smallest has been deleted update A;

    %note that velocities are sucessively deleted, and thus feasibility
    %will eventually be acheived because soon there will only be two
    %velocities left.
end
%A;

%A is now a 2-something matrix which contains the priority velocities
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Now add a pair(s) of comm constraints


S =size(C);
N= S(2);
for i = 1:N
    veloLOS(i) = norm([C(1,i),C(2,i)]);
    velorange(i) = norm([C(3,i),C(4,i)]);
    veloC(i) = norm([C(1,i)+C(3,i),C(2,i)+C(4,i)]);
end

for i = commsindx
```

```
    veloC(i) = Inf;  %if already in comms, don't evaluate in next section.
end

%if commnum <2 %if there are less than 2 bots in communication then get
% to work
%select the pair of comm velocities
for num = 1:(1-commnum)  %do this once or twice depending on how many bots
% there are
    [minC i] = min(veloC); %find the smallest pair of comm vectors
    while(minC ~= Inf)
        if veloLOS(i) ~=0
    feasible = feasiblity([A,[C(1,i);C(2,i)]]); %is it feasible?
            if (feasible)
A = [A,[C(1,i);C(2,i)]];  %if so add it to A
            end
        end

        if velorange(i) ~=0
    feasible = feasiblity([A,[C(3,i);C(4,i)]]); %is that range
% feasible?
            if (feasible)
A = [A,[C(3,i);C(4,i)]];  %if so add it to A
            end
        end

%A
veloC(i) = Inf;  %change this so a new LOS is chosen
[minC i] = min(veloC); %find the smallest comm pair
    end
end  %ends for loop
%end  %end if statement about if commnum <2

%A
% S = size(A);
% if S(2)==1;
%     feasibleVel =A;
%     return
% end
%
% %Now that we have a set of feasible vectors its time to create the final
% %motion vector!!!!
% feasibleVel = smartSum(A);
```

```
function feasibleVel = smartsum(A)
% this function takes a matrix A
% which contains a set of N 2-D vectors concatenated to form a
% 2XN matrix.  Each vector represents the desired velocity for one
% of the potential functiosn that are currently active

%  The function attempts to compute a set of two basis vectors, such
%  that any velocity vector which lies inside their convex hull would
%  simulatneously decrease all the potential functions.

% The function returns basis1 and
```

```
% basis2.  A feasible velocity is any vector beta*(alpha*basis1 +
% (1-alpha)*basis2), where alpha in [0,1] and beta in [0,infty)

%Here are two test examples
%feasible
%A= [1 1 -.1 1;1 2 -4 -1]
% infeasible
%A= [-2 1 1 -.1 1;0 1 2 -4 -1]
%----------------------------------------
% alpha =.5;
% beta = 1;


% determine number of potential fuctions
 S = size(A);
 N = S(2);

%
% figure(2)
% clf
% hold on;
% grid on;
%axis([-1 1 -1 1]);
for(i=1:N)
    %plots vectors from potentials...
    %plot([0 A(1,i)],[0 A(2,i)] ,'o-k')
    % compute the angle of each vector
    theta(i,1) = atan2(A(2,i), A(1,i))'*180/pi;
    % theta in [0,360] format
    if (theta(i)<0)
        theta(i) = 360+theta(i);
    end
end

% use column vector ...it should be one already?
theta = theta';

% now sort by increaseing theta, ind lets you reference back to actual
% vectors in A later
[theta ind] = sort(theta);
%what's going on here?
theta = theta';

% looking at change in theta as we go through list
deltaTheta = [theta(1)-theta(N); theta(2:N)- theta(1:N-1)];
%again make sure that deltaTheta in [0,360]
for(i=1:N)
    if (deltaTheta(i)<0)
        deltaTheta(i) = 360+deltaTheta(i);
    end
end

[maxDtheta, j] = max(deltaTheta);


if j == 1
    constr1 = N;
    constr2 = 1;

else
    constr1 = j;
    constr2 = j-1;
```

```
end
% remember A was not sorted!
v1 = A(:,ind(constr1));
v2 = A(:,ind(constr2));

% now must find which side of nullspaces is the good part and compute
basis1 = [-1*v1(2);v1(1)];
basis2 = [-1*v2(2); v2(1)];
%dot(v1, basis2);
%dot(v2, basis1);
% if angle greater than 90 shodul flip
if (dot(v1, basis2) <0)
    basis2 = -1*basis2;
end
if (dot(v2, basis1) <0)
    basis1 = -1*basis1;
end

% basis1;
% basis2;

%find the angular difference between the basis1 and 2
angle1 = atan2(v1(2),v1(1));
angle2 = atan2(v2(2),v2(1));
if angle1 <0
    angle1 = angle1+2*pi;
end
if angle2 <0
    angle2 = angle2+2*pi;
end

deltangle = abs(angle1 -angle2); %I'm not playing any wrap games,
%the absolute value of the difference of two positive numbers should be
% the
%angular difference between them.

if deltangle < (pi/2)  %which means that v1 and v2 are less than 90
% degrees apart
    basis1 = v1;  %if so, set the basis at the vectors themselves
    basis2 = v2;

    feasibleVel = basis1 + basis2;

%     quiver(0,0,basis1(1),basis1(2),'r')
%     quiver(0,0,basis2(1),basis2(2))
%
%     plot([0 feasibleVel(1,1)],[0 feasibleVel(2,1)],'mo-')
    return
end

mag1 = norm(basis1);
mag2 = norm(basis2);

% this plots some sample vectors for visualization purposes
%feasibleVel = beta.*(basis1*alpha + basis2*(1-alpha));  %old control law
%(which isn't the original intent either)

feasibleVel = (mag2/mag1)*basis1 + (mag1/mag2)*basis2;
% quiver(0,0,(mag2/mag1)*basis1(1),(mag2/mag1)*basis1(2),'r')
% quiver(0,0,(mag1/mag2)*basis2(1),(mag1/mag2)*basis2(2))
%
% plot([0 feasibleVel(1,1)],[0 feasibleVel(2,1)],'mo-')
```

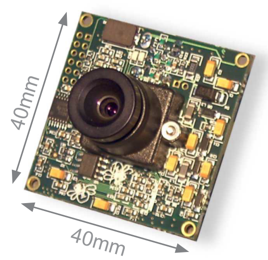
F Technical specifications for FireFly2 camera

compact, low cost IEEE-1394 digital camera

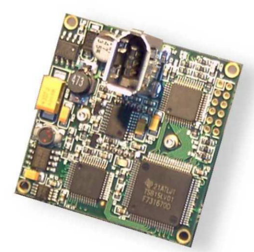
- 6 Pin IEEE-1394 interface
- 1/4" progressive scan Sony CCD
- Compact size - 40x40mm
- 640x480 uncompressed color images
- Low cost OEM board camera soulution

firefly2

POINT GREY RESEARCH



front view



back view

Firefly2 is a compact board level IEEE-1394 Digital Video Camera. Firefly2 uses a 1/4" progressive scan CCD in order to stream VGA quality color images at 30 FPS without compression. The camera is provided as a complete development kit with an IEEE-1394 interface card, cable, and image acquisition software.

Package includes:
- Firefly2 board level camera
- 4.5 meter, 6-pin, IEEE-1394 cable
- 4, 6 and 8mm focal length M12 micro lenses
- IEEE-1394 OHCI PCI Interface card
- PGRFlyCapture image acquisition and camera control C/C++ SDK

System requirements:
- Intel Pentium II or better
- Windows 2000 or XP

Camera specifications:

Imaging Device	1/4" Sony CCD (ICX098AK) Color VGA 640x480 format HAD image sensor with square pixels Progressive scan
Supported frame rates:	3.75, 7.5, 15 & 30 FPS
Supported formats:	YUV 4:1:1, YUV 4:2:2, YUV 4:4:4, and RGB 24-bit
Digital camera specification:	Version 1.04
Signal to noise ratio:	>40dB
Connector:	6-pin IEEE-1394, vertical
Power:	Through IEEE-1394, 625mW standby, 1.25 W active
Brightness:	Auto/Manual (-3dB to 33dB)
Exposure:	Auto/Manual (1/25s to 1/15000s)
Saturation:	Manual
White Balance:	Auto/Manual
Lens focal length:	4mm, 6mm or 8mm (included in the kit)
Footprint:	40 x 40mm
Weight:	12g with a micro lens

www.ptgrey.com

point grey RESEARCH

305-1847 West Broadway, Vancouver, B.C.
Canada, V6J 1Y6
T: 604-730-9937 F: 604-732-8231

Point Grey Research is a product engineering and technology company founded in January 1997. The company designs and develops computer vision technologies for commercial applications worldwide. Point Grey Research technology has been successfully used in people tracking, object tracking, modeling and dimensioning, mobile robotics, mining and many other computer vision applications.

G Programs for computer vision

```
%Obstacle Avoidence Pre Processing Function
%tic
%gets a snapshot
frame = getsnapshot(vid2);
%converts it to a black and white image
BW = frame(:,:,1) >190 & frame(:,:,3) <110;
%imshow(BW);

%lables each region and finds statistics about it
[L num] =bwlabel(BW);
stats =regionprops(L, 'Area', 'Centroid', 'ConvexHull');
idx =find([stats.Area]>80);  %filters the output so only large areas are
% shown

bw2 =ismember(L,idx); %extra stuff for plotting
%figure(2);
imshow(bw2);
hold on;
%plot( stuff for convex hull  )
%toc

pruner = 7; %pruning radius

dc = 15; %cutoff distance
%t =[0:.1:2*pi];

for n = idx %for each obstacle

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %find the sphere of Influence
    for m =1:length(stats(n).ConvexHull)
distances(m)...
=distance(stats(n).Centroid(1),stats(n).Centroid(2),stats(n).ConvexHull(m,...
1),stats(n).ConvexHull(m,2));

    end

    SOI(n) =max(distances) +dc; % sphere of influence = max distance +
% cutoff distance

% %plot(SOI(n)*cos(t)+stats(n).Centroid(1),SOI(n)*sin(t)+stats(n).Centroid
% (2),'r');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%
    clear vx; clear vy; clear Cx; clear Cy;clear angles;
    vx =stats(n).ConvexHull(:,1);
    vy =stats(n).ConvexHull(:,2);
    Cx = stats(n).Centroid(1);
    Cy = stats(n).Centroid(2);
    count =0;
    L = length(vx)-1;
    %plot(vx,vy,'r *')

    for i=[1:1:L]
%%%%%%%%%%%%%%%%%%%%
%prune out some of the excess
        if( abs(vx(i)-vx(i+1)) < pruner && abs(vy(i)-vy(i+1)) <pruner)
            vx(i) = 0;
            vy(i) =0;
        end
```

```
    end
    vx = nonzeros(vx);
    vy = nonzeros(vy);


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%
    %find the angles from the centroid to the vetexes. for each obstacle
    for g=[1:1:length(vx)]
        angles(g) = atan2(vy(g)-Cy,vx(g)-Cx);
    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%sort the angles
    anglesxy =[angles',vx,vy];

    anglesxy = sortrows(anglesxy);  %puts the angles in order, x and y
% change corispondingly
    stats(n).angles = anglesxy(:,1);

    stats(n).vx = anglesxy(:,2);
    stats(n).vy = anglesxy(:,3);

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%
    plot(vx,vy,'g *')
end
```

```
function [x, y, orient] = robotpose(vid2)
%code for the full implimentation starting with getting a snapshot and
%outputting the pose of the robot, nothing more.

%tic
frame = getsnapshot(vid2);

red = frame(:,:,2) <100 & frame(:,:,3)>150;

L =bwlabel(red);
robostats =regionpropsslick(L);  %a special function I created which only
% has Area and centroid hard coded into it
[maximus m] = max([robostats.Area]);
robostats(m).Area =0;
[maximus n] =max([robostats.Area]);

orient = atan2(robostats(n).Centroid(2) -...
robostats(m).Centroid(2),robostats(n).Centroid(1)-robostats(m).Centroid(1)...
);

x = robostats(n).Centroid(1);
y = robostats(n).Centroid(2);

%toc
%
%  figure(2);
%  imshow(red);hold on;
```

```
%  plotter = [robostats(n).Centroid(1),robostats(n).Centroid(2);
%
% robostats(n).Centroid(1)+30*cos(orreint),robostats(n).Centroid(2)+30*sin
% (orreint)];
%  plot(plotter(:,1),plotter(:,2));
%  plot(robostats(n).Centroid(1),robostats(n).Centroid(2),'o');
% %
```

```
function robostats = regionpropsslick(L);
%this was parced together from the MATLAB region props code, only I
%selected the applicable sections, notably got rid of ParseInputs which
% was
%taking the most time of any of the functions.

allStats = {'Area'
    'Centroid'
    'PixelIdxList'
    'PixelList'
    'PerimeterCornerPixelList'};

numObjs = round(double(max(L(:))));

% Initialize the stats structure array.
numStats = length(allStats);
empties = cell(numStats, numObjs);
robostats = cell2struct(empties, allStats, 1);

%Compute statistics.
%Compute Area
robostats = ComputePixelIdxList(L, robostats);

for k = 1:length(robostats)
    robostats(k).Area = size(robostats(k).PixelIdxList, 1);
end

%Compute Centroid
robostats = ComputePixelList(L, robostats);
for k = 1:length(robostats)
    robostats(k).Centroid = mean(robostats(k).PixelList,1);
end


%%%
%%% ComputePixelIdxList
%%%
function robostats = ComputePixelIdxList(L, robostats)
%   A P-by-1 matrix, where P is the number of pixels belonging to
%   the region.  Each element contains the linear index of the
%   corresponding pixel.

% Form a sparse matrix containing one column per region.  In
% column P, the location of nonzero values correspond to the
% linear indices of pixels in L that have value P.  For
% example, S(100,5) is nonzero if and only L(100) equals 5.
```

```
idx = find(L);
elementValues = L(idx);
S = sparse(idx, double(elementValues), 1);

for k = 1:length(robostats)
    robostats(k).PixelIdxList = find(S(:,k));
end


%%%
%%% ComputePixelList
%%%
function robostats = ComputePixelList(L, robostats)
%   A P-by-2 matrix, where P is the number of pixels belonging to
%   the region.  Each row contains the row and column
%   coordinates of a pixel.

robostats = ComputePixelIdxList(L, robostats);

% Loop over each column of the sparse matrix.  Finding the
% row indices of the nonzero entries in S(:,P) is equivalent
% to finding the linear indices of pixels in L that equal P.
% Convert the linear indices to subscripts and store
% the results in the pixel list.  Reverse the order of the first
% two subscripts to form x-y order.
In = cell(1,ndims(L));
for k = 1:length(robostats)
    if ~isempty(robostats(k).PixelIdxList)
        [In{:}] = ind2sub(size(L), robostats(k).PixelIdxList);
        robostats(k).PixelList = [In{:}];
        robostats(k).PixelList = robostats(k).PixelList(:,[2 1 3:end]);
    else
        robostats(k).PixelList = zeros(0,ndims(L));
    end
end
```

H Values for color segmentation

Table 6.1: Color layer intensity values for color segmentation

Selected color	Y value	U value	V value
Red	–	< 100	> 150
Blue	–	> 160	< 120
Green	> 190	–	< 110
Orange	> 190	< 80	> 120
Purple	> 190	> 145	> 135

I C program running on Koala Robot

```
/*filename: original.c */
/* reads speed commands and writes them to the wheel motors */
/* deletes the file once it is finished with it */


#include<stdio.h>

FILE * speedfile;
FILE * motorcomds;
FILE * dummy;
int speedL;
int speedR;

main()
{
while(1)
{
dummy = fopen("/work/dummy.txt", "r");

if(!dummy)
{
/*puts("an error occured while opening the file dummy");*/
}
else
{
fclose(dummy);
speedfile = fopen("/work/matlabfile.txt", "r");

if(!speedfile) /*if it can't open !0 = 1 */
{
puts("an error occured while opening the file matlabfile");
}
else
{

fscanf(speedfile, "%d\n%d", &speedL, &speedR);
fclose(speedfile);
/*remove("/work/matlabfile.txt");*/
remove("/work/dummy.txt");

motorcomds = fopen("/dev/ttyS0", "w");
fprintf(motorcomds, "D,%d,%d\r", speedL, speedR);
 printf("D,%d,%d\n", speedL,speedR);
fclose(motorcomds);

}
}


}
}
```

J Programs for physical experiments

```
%Program to Run a implimentation on the Koala robots
%nothing pretty about the code, but it works

t =0;
tfin = 60;   %the final time
tstep = .5; %the time incriment
cyclenum =0;

%Parameters
robonum =4;
velscalar = 15;
%%%%%%%%%%%%%%

%Pre allocate matrixes to imporove computational time
numberofcycles = length([0:tstep:tfin])-1;
connectedness = false([robonum,robonum,numberofcycles]);
positions = zeros([2,robonum,numberofcycles]);
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

while(t<tfin)
    clear vel;
    cyclenum =cyclenum+1;

    %Find the location of the robots
    frame = getsnapshot(vid2);

    red = frame(:,:,2) <110 & frame(:,:,3)>150;  %Knowles1 N drive
    blue = frame(:,:,2) >160 & frame(:,:,3) <120; %Knowles2 O drive
    purple = frame(:,:,1) > 190 & frame(:,:,2) >135 & frame(:,:,3) >125; %
% Knowles3 P drive
    oragne = frame(:,:,1) > 190 & frame(:,:,2) <80 & frame(:,:,3) > 120;
% %Knowles5 R drive

    %Find Red position
    L =bwlabel(red);
    robostats =regionpropsslick(L);  %a special function I created which
% only has Area and centroid hard coded into it
    [maximus m] = max([robostats.Area]);
    robostats(m).Area =0;
    [maximus n] =max([robostats.Area]);

    orient(1) = atan2(robostats(n).Centroid(2) -...
robostats(m).Centroid(2),robostats(n).Centroid(1)-robostats(m).Centroid(1)...
);

    xbots(1) = robostats(m).Centroid(1);
    ybots(1) = robostats(m).Centroid(2);

    %  figure(1);
    %  imshow(red);hold on;
    %  plotter = [robostats(n).Centroid(1),robostats(n).Centroid(2);
    %
% robostats(n).Centroid(1)+30*cos(orient(1)),robostats(n).Centroid(2)+30*s
% in(orient(1))];
    %  plot(plotter(:,1),plotter(:,2));
    %  plot(robostats(n).Centroid(1),robostats(n).Centroid(2),'o');
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    L =bwlabel(blue);
    robostats =regionpropsslick(L);  %a special function I created which
% only has Area and centroid hard coded into it
```

```
    [maximus m] = max([robostats.Area]);
    robostats(m).Area =0;
    [maximus n] =max([robostats.Area]);

    orient(2) = atan2(robostats(n).Centroid(2) -...
robostats(m).Centroid(2),robostats(n).Centroid(1)-robostats(m).Centroid(1)...
);

    xbots(2) = robostats(m).Centroid(1);
    ybots(2) = robostats(m).Centroid(2);

    %  figure(2);
    %  imshow(blue);hold on;
    %  plotter = [robostats(n).Centroid(1),robostats(n).Centroid(2);
    %
% robostats(n).Centroid(1)+30*cos(orient(2)),robostats(n).Centroid(2)+30*s
% in(orient(2))];
    %  plot(plotter(:,1),plotter(:,2));
    %  plot(robostats(n).Centroid(1),robostats(n).Centroid(2),'o');
    % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    L =bwlabel(purple);
    robostats =regionpropsslick(L);  %a special function I created which
% only has Area and centroid hard coded into it
    [maximus m] = max([robostats.Area]);
    robostats(m).Area =0;
    [maximus n] =max([robostats.Area]);

    orient(3) = atan2(robostats(n).Centroid(2) -...
robostats(m).Centroid(2),robostats(n).Centroid(1)-robostats(m).Centroid(1)...
);

    xbots(3) = robostats(m).Centroid(1);
    ybots(3) = robostats(m).Centroid(2);

    %  figure(3);
    %  imshow(purple);hold on;
    %  plotter = [robostats(n).Centroid(1),robostats(n).Centroid(2);
    %
% robostats(n).Centroid(1)+30*cos(orient(3)),robostats(n).Centroid(2)+30*s
% in(orient(3))];
    %  plot(plotter(:,1),plotter(:,2));
    %  plot(robostats(n).Centroid(1),robostats(n).Centroid(2),'o');
    %
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5
%     L =bwlabel(oragne);
%     robostats =regionpropsslick(L);  %a special function I created which
% only has Area and centroid hard coded into it
%     [maximus m] = max([robostats.Area]);
%     robostats(m).Area =0;
%     [maximus n] =max([robostats.Area]);
%
%     orient(4) = atan2(robostats(n).Centroid(2) -
% robostats(m).Centroid(2),robostats(n).Centroid(1)-robostats(m).Centroid(
% 1));
%     xbots(4) = robostats(m).Centroid(1);
%     ybots(4) = robostats(m).Centroid(2);

    %  figure(4);
    %  imshow(oragne);hold on;
    %  plotter = [robostats(n).Centroid(1),robostats(n).Centroid(2);
    %
% robostats(n).Centroid(1)+30*cos(orient(4)),robostats(n).Centroid(2)+30*s
```

```
% in(orient(4))];
    %  plot(plotter(:,1),plotter(:,2));
    %  plot(robostats(n).Centroid(1),robostats(n).Centroid(2),'o');
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    positions(1,:,cyclenum) = xbots;
    positions(2,:,cyclenum) = ybots;  %positions is a record of all the
% positions overtime

    %find the velocity for each robot
    for r= 1:robonum
[vel(:,r), connectedness] = velcontroller(r, cyclenum, xbots,...
ybots, connectedness, stats, idx);


        vel(:,r) = jacob(vel(1,r),vel(2,r), orient(r));
    end

    vel = int8(velscalar*vel); %this maxes out at 127 and -128

    flag =true;
    while(flag)
        dummy = fopen('N:\\dummy.txt' ,'r');
if (dummy < 0) %if dummy isn't there

            speedfile = fopen('N:\\matlabfile.txt', 'w');
    fprintf(speedfile, '%d\n%d', vel(1,1),vel(2,1));  %this is for
% c code
            fclose(speedfile);

            writedummy = fopen('N:\\dummy.txt', 'w');
            fclose(writedummy);
            flag = false;
        end
    end

    flag =true;
    while(flag)
        dummy = fopen('O:\\dummy.txt' ,'r');
if (dummy < 0) %if dummy isn't there

            speedfile = fopen('O:\\matlabfile.txt', 'w');
    fprintf(speedfile, '%d\n%d', vel(1,2),vel(2,2));  %this is for
% c code
            fclose(speedfile);

            writedummy = fopen('O:\\dummy.txt', 'w');
            fclose(writedummy);
            flag = false;
        end
    end

    flag =true;
    while(flag)
        dummy = fopen('P:\\dummy.txt' ,'r');
if (dummy < 0) %if dummy isn't there

            speedfile = fopen('P:\\matlabfile.txt', 'w');
    fprintf(speedfile, '%d\n%d', vel(1,3),vel(2,3));  %this is for
% c code
            fclose(speedfile);
```

```
            writedummy = fopen('P:\\dummy.txt', 'w');
            fclose(writedummy);
            flag = false;
        end
    end

%     flag =true;
%     while(flag)
%   dummy = fopen('R:\\dummy.txt' ,'r');
%   if (dummy < 0) %if dummy isn't there
%
%       speedfile = fopen('R:\\matlabfile.txt', 'w');
%       fprintf(speedfile, '%d\n%d', vel(1,4),vel(2,4)); %this is
% for c code
%       fclose(speedfile);
%
%       writedummy = fopen('R:\\dummy.txt', 'w');
%       fclose(writedummy);
%       flag = false;
%   end
%     end

    t =t+tstep;
end

vel = zeros([2,4]);
vel = int8(vel);

speedfile = fopen('N:\\matlabfile.txt', 'w');
fprintf(speedfile, '%d\n%d', vel(1,1),vel(2,1));  %this is for c code
fclose(speedfile);

writedummy = fopen('N:\\dummy.txt', 'w');
fclose(writedummy);

speedfile = fopen('O:\\matlabfile.txt', 'w');
fprintf(speedfile, '%d\n%d', vel(1,2),vel(2,2));  %this is for c code
fclose(speedfile);

writedummy = fopen('O:\\dummy.txt', 'w');
fclose(writedummy);

speedfile = fopen('P:\\matlabfile.txt', 'w');
fprintf(speedfile, '%d\n%d', vel(1,3),vel(2,3));  %this is for c code
fclose(speedfile);

writedummy = fopen('P:\\dummy.txt', 'w');
fclose(writedummy);

% speedfile = fopen('R:\\matlabfile.txt', 'w');
% fprintf(speedfile, '%d\n%d', vel(1,4),vel(2,4));  %this is for c code
% fclose(speedfile);
%
% writedummy = fopen('R:\\dummy.txt', 'w');
% fclose(writedummy);
```

```
function [vel, connectedness] = velcontroller(r, cyclenum, xbots, ybots,...
connectedness, stats, idx)
...

%Tom Dunbar
%5FEB
%this program sets up the velocities necessary to calculate a robot's
%new velocity and then passes the information on to


%Parameters%%%%%%%%%%%%%%%%%%%%%%%%
robonum = 4; %the number of robots in the simulation
gscale =20;  %goal scaler, note that 1/gscale*gdx
oscale =25; %obstacle avoidence scaler, oscale*oadx
%maxvel = 5; %maximum velocity
minsep = 50; %this is devided by 10;
gx = 300;
gy = 220;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


P = [0;0];
C = zeros(4,robonum);

%Goal velocity%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
gdx = 1/gscale*(gx-xbots(r));
gdy = 1/gscale*(gy-ybots(r));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (sqrt((gx-xbots(r))^2+ (gy-ybots(r))^2) < 10 )  %if robot is near the
% goal, do nothing,
    'robot at goal'
    [vel, connectedness] =  paracontroller(P,C,r,cyclenum, connectedness);
% %just so the connectedness matrix gets filled in
    return
else
    P(:,1) = [gdx;gdy];
end

for o = idx %for each obstacle

    %Obstacle avoidance%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [oadx oady] = Bishopobstacle(xbots(r), ybots(r), o, stats);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if(sqrt(oadx^2+oady^2) ~= 0)   %if there is a vector, pass it to the
% controller
        P(:,2) = [oadx;oady].*oscale;
    end

    %LOS stuff%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Co = zeros(4,robonum);
    for q =1:robonum %check LOS with each robot

if q ~= r  %the robot will always have LOS with itself...

            A = [stats(o).vx';stats(o).vy'];

            xB = xbots(q);
            yB = ybots(q);

    Ab(1,:) = A(1,:) - xB;   %translating the obstacle so B is at
```

```
% the origin, do it can be checked by LOSchecker
            Ab(2,:) = A(2,:) - yB;

            xtrans = xbots(r) -xB;
            ytrans = ybots(r) -yB;

    [flag, outside, gammab, gammat] = LOSchecker(xtrans,ytrans,...
Ab);


            if (flag)
[LOSdx,LOSdy] = LOSvelf(xtrans,ytrans, gammab, gammat,...
outside);

                Co(1,q) = LOSdx;
                Co(2,q) = LOSdy;
            end
end  %end if statement
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    end %end for each robot q


    for c = 1:robonum
        if (abs(C(1,c)) < abs(Co(1,c)))
    C(1,c) = Co(1,c); %this replaces a previous LOS vector
        end
        if (abs(C(2,c)) < abs(Co(2,c)))
            C(2,c) = Co(2,c);
        end
    end

end  %end for each obstacle

%Range part%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for q = 1:robonum %for each robot
    if q~=r
    dist = sqrt((xbots(q)-xbots(r))^2 + (ybots(q)-ybots(r))^2); %dist = 0
% if they are the same

    xunit = (xbots(q) - xbots(r))/dist;   %must be written this way to
% ensure vector is pointing the correct way.
    yunit = (ybots(q) - ybots(r))/dist;

    if (dist && dist < (minsep))  %dist must be nonzero and less than
% minsep
        delta =.01;
        h =.556929;
xsig =1.1229*h;  %these values based off of a valley at 2.5
        c = 7.05538;
%mexhat
dist = dist/minsep; %this changes distance to a value from 0 to 1

zplus  =...
(1/(2*pi*xsig))*c*(h-(dist+delta)^2/xsig^2)*exp(-1*(dist+delta)^2/(2*xsig^...
2));

zminus =...
(1/(2*pi*xsig))*c*(h-(dist-delta)^2/xsig^2)*exp(-1*(dist-delta)^2/(2*xsig^...
2));

        dz = (zplus-zminus)/(2*delta);
```

```
        P(1,end+1) = dz*xunit;
P(2,end+1) = dz*yunit; %if the distance is less than minimum
% seperation,
    else
        dz = Impsemi(dist/10);
C(3,q) = dz*xunit; %velocity times unit vector
        C(4,q) = dz*yunit;
    end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   [M N] = size(P);
%   for p = 1:N
%       quiver(xbots(r),ybots(r),P(1,p),P(2,p),'b')
%   end
%     [M N] = size(C);
%     for c=1:N
%   quiver(xbots(r),ybots(r),C(1,c),C(2,c),'g')
%   quiver(xbots(r),ybots(r),C(3,c),C(4,c),'r')
%     end
%     P
%     C

[vel, connectedness] =  paracontroller(P,C,r,cyclenum, connectedness);

%figure(1)
%quiver(xbots(r),ybots(r),feasibleVel(1),feasibleVel(2), 'm')

%   dmag = norm(feasibleVel);  %this stuff should be taken care of
% in
%   %the int8 convertion
%
%   if (dmag >maxvel)
%       feasibleVel = maxvel*feasibleVel/dmag;
%   end
%
%vel =feasibleVel.*tstep;
```

```
vel = zeros([2,4]);
vel = int8(vel);

speedfile = fopen('N:\\matlabfile.txt', 'w');
fprintf(speedfile, '%d\n%d', vel(1,1),vel(2,1));  %this is for c code
fclose(speedfile);

writedummy = fopen('N:\\dummy.txt', 'w');
fclose(writedummy);

speedfile = fopen('O:\\matlabfile.txt', 'w');
fprintf(speedfile, '%d\n%d', vel(1,2),vel(2,2));  %this is for c code
fclose(speedfile);

writedummy = fopen('O:\\dummy.txt', 'w');
fclose(writedummy);
```

```
speedfile = fopen('P:\\matlabfile.txt', 'w');
fprintf(speedfile, '%d\n%d', vel(1,3),vel(2,3));  %this is for c code
fclose(speedfile);

writedummy = fopen('P:\\dummy.txt', 'w');
fclose(writedummy);

% speedfile = fopen('R:\\matlabfile.txt', 'w');
% fprintf(speedfile, '%d\n%d', vel(1,4),vel(2,4));  %this is for c code
% fclose(speedfile);
%
% writedummy = fopen('R:\\dummy.txt', 'w');
% fclose(writedummy);
```

```
function [vel] = jacob(dx,dy, theta)

R = 8.5; %in cm
W = 32;  %in cm taken from appendix as the width
D = 13.5;% in cm, as the distance from the middle of the wheel to the
% control point,
%note that this is inaccurate because the control point will vary as the
%image processor calculates the centroid of the small dot.

J = [ R/2*cos(theta)-R*D/W* sin(theta), R/2*cos(theta)+R*D/W*sin(theta);
      R/2*sin(theta)+R*D/W* cos(theta), R/2*sin(theta)-R*D/W*cos(theta)];

vel = J^(-1)*[dx;dy];
```

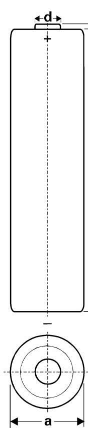
K Technical specifications for replacement batteries

VARTA

VH 4000 4/3A

Rechargeable Ni-MH Cylindrical

Data Sheet



Type Number:	55140	
System:	Nickel Metal Hydride/ KOH Electrolyte	
Nominal Voltage [V]:	1.2	
Nominal Capacity C [mAh]:	3600	
Typical Capacity C [mAh]: at 800mA / 1.00V	4000	
Weight, approx. [g]	55.0	
Dimensions [mm]:	**min.**	**max.**
Diameter [a]:	16.0	17.0
Height [b]:	65.5	67.5
Shoulder Height [c]	65.2	67.7
Cap diameter [d]	7.5	8.5
Temperature Ranges [°C]	**min.**	**max.**
Storage: less than 30 days	-20	50
less than 90 days	-20	40
less than 1 year	-20	30
Discharge:	-20	60
Charge:	0	45
Charging Method:		
Normal Charging:	370mA for 14 – 16h	
Accelerated Charging (20°C): Time controlled, voltage control recommended	1100mA for 4.75h	
Fast Charging: (20°C)	3000mA *	
Trickle Charging:	pulsed recommended	
Charge Retention [%] at 20°C: Capacity available after 1 month Storage at 20°C (cell was fast charged)	min. 60%	
Impedance [mOhm]: at charged cells (5 cycles), 20°C, AC: 1kHz, (IEC 61951-2)	max. 60	
Typical Capacities [mAh]:		
at 800mA / 1.00V	4000	
at 2A / 1.00V	3700	
at 4A / 1.00V	3600	
Max. Discharge Current (cont.) [mA]:	5000	
Life Expectancy (typical):		
IEC Cycle:	>500 Cycles	

* (dT/ dt, -dV)

Capacities based on normal charging

L Programs for Simulations

```
%workspace creater

%22Jan
clear
clc
global stats
robonum = 5; %the number of robots in the simulation

figure(1)
clf;
hold on;
axis([0 300 0 300])

for k=[1:1:3]  % get vertices of obstacles
    xys=ginput;
    vx(k) = xys(1,1);
    vy(k) = xys(1,2);
    plot(vx(k),vy(k), 'g *');
end

fill(vx,vy, 'k')

sarea=0;
scx =0;
scy =0;

n= (length(vx));

for i =1:1:(n-1)
    sarea = sarea + (vx(i)*vy(i+1) - vx(i+1)*vy(i));
    scx = scx + (vx(i)+vx(i+1))*(vx(i)*vy(i+1) - vx(i+1)*vy(i));
    scy = scy + (vy(i)+vy(i+1))*(vx(i)*vy(i+1) - vx(i+1)*vy(i));
end

area = abs(.5*(sarea + vx(n)*vy(1)-vx(1)*vy(n)));
Cx = abs(1/(6*area)* (scx+ (vx(n)+vx(1))*(vx(n)*vy(1) - vx(1)*vy(n))));
Cy = abs(1/(6*area)* (scy+ (vy(n)+vy(1))*(vx(n)*vy(1) - vx(1)*vy(n))));

plot(Cx,Cy, 'r d');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%
%find the angles from the centroid to the vetexes. for each obstacle
for g=[1:1:length(vx)]
    angles(g) = atan2(vy(g)-Cy,vx(g)-Cx);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%sort the angles
anglesxy =[angles',vx',vy'];

anglesxy = sortrows(anglesxy); %puts the angles in order, x and y change
% corispondingly
stats(1).angles = anglesxy(:,1);

stats(1).vx = anglesxy(:,2);
stats(1).vy = anglesxy(:,3);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%

stats(1).Area = area;
```

```
stats(1).Centroid(1) = Cx;
stats(1).Centroid(2) = Cy;


%set up the robot positions

for k=[1:1:robonum]
    xys=ginput;
    xbots(k) = xys(1,1);
    ybots(k) = xys(1,2);
    plot(xbots(k),ybots(k), 'r >'); %get the robot positions
end

for k=[1:1:robonum]
    xys=ginput;
    gx(k) = xys(1,1);
    gy(k) = xys(1,2);
    plot(gx(k),gy(k), 'b *');   %get the goal positions
end
```

```
%clear
robonum = 15;
minsep = 10;
maxcom = 50;

figure(1)
clf;
hold on;
axis([0 500 0 500])
%w=2
for w = 28:28
    robonum = 15;
    minsep = 10;
    maxcom = 50;
    %   figure(1)
    % clf;
    % hold on;
    % axis([0 500 0 500])
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %set up goal space
    flag =true;
    count =2;
    while(flag & count>=2)
        gxy = rand(2,robonum)*100+400;
        for r =1:robonum
            count =0;
            for i = 1:robonum

dist = sqrt((gxy(1,i) - gxy(1,r))^2 +(gxy(2,i) -...
gxy(2,r))^2);


                if minsep > dist
                    flag =false;
```

```
                break
            end

            if dist <= maxcom
                count = count+1;
            end
        end
    end
end

plot(gxy(1,:),gxy(2,:), '*')
gx = gxy(1,:);
gy = gxy(2,:);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%set up robot positions
xbots = rand(1,robonum)*200;
ybots = rand(1,robonum)*200;
plot(xbots,ybots, 'r .')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%set up triangle obstacle
%for k=[1:1:3]  % get vertices of obstacles

xys=ginput;
vx = xys(:,1);
vy = xys(:,2);
plot(vx,vy, 'g *');
%end

fill(vx,vy, 'k')

sarea=0;
scx =0;
scy =0;

n= (length(vx));

for i =1:1:(n-1)
    sarea = sarea + (vx(i)*vy(i+1) - vx(i+1)*vy(i));
    scx = scx + (vx(i)+vx(i+1))*(vx(i)*vy(i+1) - vx(i+1)*vy(i));
    scy = scy + (vy(i)+vy(i+1))*(vx(i)*vy(i+1) - vx(i+1)*vy(i));
end

area = abs(.5*(sarea + vx(n)*vy(1)-vx(1)*vy(n)));
Cx = abs(1/(6*area)* (scx+ (vx(n)+vx(1))*(vx(n)*vy(1) -...
vx(1)*vy(n))));

Cy = abs(1/(6*area)* (scy+ (vy(n)+vy(1))*(vx(n)*vy(1) -...
vx(1)*vy(n))));


plot(Cx,Cy, 'r d');


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%
%find the angles from the centroid to the vetexes. for each obstacle
for g=[1:1:length(vx)]
    angles(g) = atan2(vy(g)-Cy,vx(g)-Cx);
```

```
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %sort the angles
    anglesxy =[angles',vx,vy];

    anglesxy = sortrows(anglesxy);  %puts the angles in order, x and y
% change corispondingly
    stats(1).angles = anglesxy(:,1);

    stats(1).vx = anglesxy(:,2);
    stats(1).vy = anglesxy(:,3);
    stats(1).Area = area;
    stats(1).Centroid(1) = Cx;
    stats(1).Centroid(2) = Cy;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%
    %set up square obstacle
    %for k=[1:1:4]  % get vertices of obstacles
    clear xys, vx,vy
    xys=ginput;
    vx = xys(:,1);
    vy = xys(:,2);
    plot(vx,vy, 'g *');
    %end

    fill(vx,vy, 'k')

    sarea=0;
    scx =0;
    scy =0;

    n= (length(vx));

    for i =1:1:(n-1)
        sarea = sarea + (vx(i)*vy(i+1) - vx(i+1)*vy(i));
        scx = scx + (vx(i)+vx(i+1))*(vx(i)*vy(i+1) - vx(i+1)*vy(i));
        scy = scy + (vy(i)+vy(i+1))*(vx(i)*vy(i+1) - vx(i+1)*vy(i));
    end

    area = abs(.5*(sarea + vx(n)*vy(1)-vx(1)*vy(n)));
    Cx = abs(1/(6*area)* (scx+ (vx(n)+vx(1))*(vx(n)*vy(1) -...
vx(1)*vy(n))));

    Cy = abs(1/(6*area)* (scy+ (vy(n)+vy(1))*(vx(n)*vy(1) -...
vx(1)*vy(n))));


    plot(Cx,Cy, 'r d');


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%
    %find the angles from the centroid to the vetexes. for each obstacle
    for g=[1:1:length(vx)]
        angles(g) = atan2(vy(g)-Cy,vx(g)-Cx);
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    %sort the angles
    anglesxy =[angles',vx,vy];

    anglesxy = sortrows(anglesxy);  %puts the angles in order, x and y
% change corispondingly
    stats(2).angles = anglesxy(:,1);

    stats(2).vx = anglesxy(:,2);
    stats(2).vy = anglesxy(:,3);
    stats(2).Area = area;
    stats(2).Centroid(1) = Cx;
    stats(2).Centroid(2) = Cy;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%

    filena = ['workspace',num2str(w)];
    save(filena,'xbots','ybots','stats','gx','gy')
    %clear
end
```

```
function [connectedness, neargoal, positions] =...
fullsim(xbots,ybots,stats,gx,gy);

%Tom Dunbar
%24MAR

%run workspace creator


%Parameters%%%%%%%%%%%%%%%%%%%%%%%%
robonum =15;
tfin = 500;  %the final time
tstep = .5; %the time incriment
gscale =20; %goal scaler, note that 1/gscale*gdx
oscale =25; %obstacle avoidence scaler, oscale*oadx
Lscale = 50;  %LOS scale
maxvel = 5; %maximum velocity
minsep =10;   %minimum seperation between bots
goalcut = 20/gscale; %maximum distance until we say the robot is near
% enough to the goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


t =0;
cyclenum =1;
%Pre allocate matrixes to imporove computational time
numberofcycles = length([0:tstep:tfin]);
connectedness = false([robonum,robonum,numberofcycles]);
positions = zeros([2,robonum,numberofcycles]);
neargoal = numberofcycles*ones([1 robonum]);
%%%%

positions(1,:,cyclenum) = xbots;  %these come from the workspacecreator
```

```
positions(2,:,cyclenum) = ybots;

connectedness(:,:,1) = connections(positions(:,:,1),stats);
% upper = triu(connects);
% lower = tril(connects);
% chec = upper & lower';
% connectedness(:,:,cyclenum) = chec;

while(t<tfin)

    for r = 1:robonum %for each robot
        P=[];
        C = zeros(4,robonum);


%Goal velocity%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        gdx = 1/gscale*(gx(r)-xbots(r));
        gdy = 1/gscale*(gy(r)-ybots(r));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (~(gdx || gdy))  %if both equal zero, the robot is at the goal
    %'robot at goal'
    P(:,1) = [.0001;.0001];  %this stops an error when the robot
% is *exactly* at the goal which never happens
        else
    P(:,1) = [gdx;gdy];  %a goal will always exist except if the
% robots reach exactly the goal.
        end

        if norm([gdx;gdy]) < goalcut
    if cyclenum < neargoal(r)  %this returns a vector of when the
% robot reaches near its goal, if it never gets near it,
neargoal(r) = cyclenum; %the vector value is the total
% number of cycles.
            end
        end

for o = 1:length(stats) %for each obstacle

    %Obstacle avoidance%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            [oadx oady] = Bishopobstacle(xbots(r), ybots(r), o, stats);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if(sqrt(oadx^2+oady^2) ~= 0)   %if there is a vector, pass it
% to the controller
                P(:,end+1) = [oadx;oady].*oscale;
            end

    %LOS stuff%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        for q =1:robonum %check LOS with each robot
if q ~= r  %the robot will always have LOS with itself
            A = [stats(1).vx';stats(1).vy'];

                    xB = xbots(q);
                    yB = ybots(q);

    Ab(1,:) = A(1,:) - xB;   %translating the obstacle so
% B is at the origin, do it can be checked by LOSchecker
                    Ab(2,:) = A(2,:) - yB;

                    xtrans = xbots(r) -xB;
                    ytrans = ybots(r) -yB;
```

```
    [outside, sanctum, gammab, gammat] =...
                LOSconnect(xtrans,ytrans,Ab); %is LOS needed for these robots?
    if (~sanctum) %if it is NOT in the inner sanctum or
% outside the occlusion lines
%then LOS needs to be evaluated.
                        connxr = sum(connectedness(r,:,cyclenum)) -1;
connxq = sum(connectedness(q,:,cyclenum)) -1;
% %minus one since it will always have conx with itself
if connxr < connxq %if r has less connextions than
% q r needs to move
    [LOSdx,LOSdy] = LOSvelf(xtrans,ytrans, gammab,...
gammat, outside);

    %P(:,end+1) = [LOSdx; LOSdy].*Lscale;
    %quiver(xbots(r),ybots(r),LOSdx,LOSdy, 'm')
                if norm([LOSdx, LOSdy]) > norm([C(1,q),C(2,q)])
                    C(1,q) = LOSdx*Lscale;  %if this is a
                    % larger LOS vector use it
                    C(2,q) = LOSdy*Lscale;
                end
            end
            %other case
            if connxr == connxq  % or if they have equal
                % connections, the lower numbered robot needs to move
                if r < q
                    [LOSdx,LOSdy] = LOSvelf(xtrans,ytrans,...
                        gammab, gammat, outside);

                    %P(:,end+1) = [LOSdx; LOSdy].*Lscale;

                    % %quiver(xbots(r),ybots(r),LOSdx*Lscale,LOSdy, 'm')
                    if norm([LOSdx, LOSdy]) > norm([C(1,q),C(2,q)])
                        C(1,q) = LOSdx*Lscale;   %if this is a
                        % larger LOS vector use it
                        C(2,q) = LOSdy*Lscale;
                    end
                end
            end
        end
end  %end if q~=r statement
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    end %end for each robot q
end  %end for each obstacle

%Range part%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for q = 1:robonum %for each robot
            if r ~=q
dist = sqrt((xbots(q)-xbots(r))^2 + (ybots(q)-ybots(r))..
        ^2); %dist = 0 if they are the same

xunit = (xbots(q) - xbots(r))/dist;   %must be written
% this way to ensure vector is pointing the correct way.
                yunit = (ybots(q) - ybots(r))/dist;

if (dist < minsep)  %dist must be nonzero
    dist = dist/minsep;  %on a scale from 0 to 1 how small
% is the distance compared to minsep
                    delta =.01;
                    h =.556929;
                    xsig =1.1229*h;
                    c = 7.05538;
```

```
    %mexhat
    zplus  =...
(1/(2*pi*xsig))*c*(h-(dist+delta)^2/xsig^2)*exp(-1*(dist+delta)^2/(2*xsig^...
2));

    zminus =...
(1/(2*pi*xsig))*c*(h-(dist-delta)^2/xsig^2)*exp(-1*(dist-delta)^2/(2*xsig^...
2));

                    dz = (zplus-zminus)/(2*delta);
    P(:,end+1) = [xunit; yunit].*dz;  %if the distance is
% less than minimum seperation,
                else
                    dz = Improvedsemicont(dist);
    C(3,q) = dz*xunit; %velocity times unit vector
                    C(4,q) = dz*yunit;
                end
            end
        end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   if r ==1
%       [M N] = size(P);
%       for u = 1:N
%   quiver(xbots(r),ybots(r),P(1,u),P(2,u),'k')
%       end
%       [M N] = size(C);
%       for c=1:N
%   quiver(xbots(r),ybots(r),C(1,c),C(2,c),'g')
%   quiver(xbots(r),ybots(r),C(3,c),C(4,c),'r')
%       end
%   end
%   P
%   C
commnum = sum(connectedness(r,:,cyclenum))-1; %minus one since it
% will have coms with itself
        commsindx = find(connectedness(r,:,cyclenum));

        A =  paracontrollerupdate(P,C, commsindx, commnum);

        N = size(A);
        if N(2)==1;
            feasibleVel =A;
        else
    %A
            feasibleVel = smartSum(A);
        end

        dmag = norm(feasibleVel);

        if (dmag >maxvel)
            feasibleVel = maxvel*feasibleVel/dmag;
        end

        dvect(:,r) =feasibleVel.*tstep;

    end
    xbots = xbots+dvect(1,:);
    ybots = ybots+dvect(2,:);

    cyclenum = cyclenum+1;
    positions(1,:,cyclenum) = xbots;
```

```
    positions(2,:,cyclenum) = ybots;

    connectedness(:,:,cyclenum) =...
connections(positions(:,:,cyclenum),stats);

    t =t+tstep;
end
```

```
function [connectedness, neargoal, positions] =...
fullsimcomms(xbots,ybots,stats,gx,gy);

%Tom Dunbar
%24MAR

%run workspace creator


%Parameters%%%%%%%%%%%%%%%%%%%%%%%
robonum =15;
tfin = 500;  %the final time
tstep = .5; %the time incriment
gscale =20; %goal scaler, note that 1/gscale*gdx
oscale =25; %obstacle avoidence scaler, oscale*oadx
Lscale = 50;  %LOS scale
maxvel = 5; %maximum velocity
minsep =10;   %minimum seperation between bots
goalcut = 20/gscale; %maximum distance until we say the robot is near
% enough to the goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


t =0;
cyclenum =1;
%Pre allocate matrixes to imporove computational time
numberofcycles = length([0:tstep:tfin]);
connectedness = false([robonum,robonum,numberofcycles]);
positions = zeros([2,robonum,numberofcycles]);
neargoal = numberofcycles*ones([1 robonum]);
%%%%

positions(1,:,cyclenum) = xbots;  %these come from the workspacecreator
positions(2,:,cyclenum) = ybots;

connectedness(:,:,1) = connections(positions(:,:,1),stats);
% upper = triu(connects);
% lower = tril(connects);
% chec = upper & lower';
% connectedness(:,:,cyclenum) = chec;

while(t<tfin)

    for r = 1:robonum %for each robot
        P=[];
        C = zeros(4,robonum);
```

```
%Goal velocity%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        gdx = 1/gscale*(gx(r)-xbots(r));
        gdy = 1/gscale*(gy(r)-ybots(r));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        if norm([gdx;gdy]) < goalcut
    if cyclenum < neargoal(r)  %this returns a vector of when the
% robot reaches near its goal, if it never gets near it,
neargoal(r) = cyclenum; %the vector value is the total
% number of cycles.
            end
        end

for o = 1:length(stats) %for each obstacle

    %Obstacle avoidance%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            [oadx oady] = Bishopobstacle(xbots(r), ybots(r), o, stats);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if(sqrt(oadx^2+oady^2) ~= 0)   %if there is a vector, pass it
% to the controller
                P(:,end+1) = [oadx;oady].*oscale;
            end

    %LOS stuff%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        for q =1:robonum %check LOS with each robot
if q ~= r  %the robot will always have LOS with itself...
            A = [stats(1).vx';stats(1).vy'];

                    xB = xbots(q);
                    yB = ybots(q);

    Ab(1,:) = A(1,:) - xB;   %translating the obstacle so
    % B is at the origin, do it can be checked by LOSchecker
                    Ab(2,:) = A(2,:) - yB;

                    xtrans = xbots(r) -xB;
                    ytrans = ybots(r) -yB;

    [outside, sanctum, gammab, gammat] =...
 LOSconnect(xtrans,ytrans,Ab); %is LOS needed for these robots?
    if (~sanctum) %if it is NOT in the inner sanctum or
            % outside the occlusion lines
%then LOS needs to be evaluated.
                        connxr = sum(connectedness(r,:,cyclenum)) -1;
connxq = sum(connectedness(q,:,cyclenum)) -1;
            % %minus one since it will always have conx with itself
if connxr < connxq %if r has less connextions than
                            % q r needs to move
    [LOSdx,LOSdy] = LOSvelf(xtrans,ytrans, gammab,...
                            gammat, outside);

    %P(:,end+1) = [LOSdx; LOSdy].*Lscale;
    %quiver(xbots(r),ybots(r),LOSdx,LOSdy, 'm')
    if norm([LOSdx, LOSdy]) > norm([C(1,q),C(2,q)]);
C(1,q) = LOSdx*Lscale;  %if this is a
                % larger LOS vector use it
                            C(2,q) = LOSdy*Lscale;
                        end
                    end
```

```
%other case
if connxr == connxq  % or if they have equal
                % connections, the lower numbered robot needs to move
                            if r < q
[LOSdx,LOSdy] = LOSvelf(xtrans,ytrans,...
                                gammab, gammat, outside);

%P(:,end+1) = [LOSdx; LOSdy].*Lscale;

                % %quiver(xbots(r),ybots(r),LOSdx*Lscale,LOSdy, 'm')
if norm([LOSdx, LOSdy]) > norm([C(1,q),C(2,q)]);
    C(1,q) = LOSdx*Lscale;   %if this is a
                    % larger LOS vector use it
                                    C(2,q) = LOSdy*Lscale;
                                end
                            end
                        end
                    end
end  %end if q~=r statement
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    end %end for each robot q
end  %end for each obstacle

%Range part%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for q = 1:robonum %for each robot
            if r ~=q
dist = sqrt((xbots(q)-xbots(r))^2 +
 (ybots(q)-ybots(r))^2); %dist = 0 if they are the same

xunit = (xbots(q) - xbots(r))/dist;   %must be written
% this way to ensure vector is pointing the correct way.
                yunit = (ybots(q) - ybots(r))/dist;

if (dist < minsep)  %dist must be nonzero
    dist = dist/minsep;  %on a scale from 0 to 1 how small
% is the distance compared to minsep
                    delta =.01;
                    h =.556929;
                    xsig =1.1229*h;
                    c = 7.05538;
    %mexhat
    zplus  =...
(1/(2*pi*xsig))*c*(h-(dist+delta)^2/xsig^2)*exp(-1*(dist+delta)^2/(2*xsig^...
2));

    zminus =...
(1/(2*pi*xsig))*c*(h-(dist-delta)^2/xsig^2)*exp(-1*(dist-delta)^2/(2*xsig^...
2));

                    dz = (zplus-zminus)/(2*delta);
    P(:,end+1) = [xunit; yunit].*dz;  %if the distance is
% less than minimum seperation,
                else
                    dz = Improvedsemicont(dist);
    C(3,q) = dz*xunit; %velocity times unit vector
                    C(4,q) = dz*yunit;
                end
            end
        end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   if r ==1
```

```
%       [M N] = size(P);
%       for u = 1:N
%   quiver(xbots(r),ybots(r),P(1,u),P(2,u),'k')
%       end
%       [M N] = size(C);
%       for c=1:N
%   quiver(xbots(r),ybots(r),C(1,c),C(2,c),'g')
%   quiver(xbots(r),ybots(r),C(3,c),C(4,c),'r')
%       end
%   end
%   P
%   C
commnum = sum(connectedness(r,:,cyclenum))-1; %minus one since it
% will have coms with itself
        commsindx = find(connectedness(r,:,cyclenum));

        A =  paracontrollerupdate(P,C, commsindx, commnum);

        feasible = feasiblity([A,[gdx;gdy]]);
if (feasible)  %this is what seperates this from the regular full
% sim
    A(:,end+1) = [gdx;gdy];  %only after we have a feasible comm
% vector, is it written to A
        end

        N = size(A);
        if N(2)==1;
            feasibleVel =A;
        else
    %A
            feasibleVel = smartSum(A);
        end

        dmag = norm(feasibleVel);

        if (dmag >maxvel)
            feasibleVel = maxvel*feasibleVel/dmag;
        end

        dvect(:,r) =feasibleVel.*tstep;

    end
    xbots = xbots+dvect(1,:);
    ybots = ybots+dvect(2,:);

    cyclenum = cyclenum+1;
    positions(1,:,cyclenum) = xbots;
    positions(2,:,cyclenum) = ybots;

    connectedness(:,:,cyclenum) =...
connections(positions(:,:,cyclenum),stats);

    t =t+tstep;
end
```

```
%function [connectedness, neargoal, positions] =
%fullsimrestric(xbots,ybots,stats,gx,gy);
%Tom Dunbar
%24MAR

%run workspace creator


%Parameters%%%%%%%%%%%%%%%%%%%%%%%
robonum =3;
tfin = 50;  %the final time
tstep = .5; %the time incriment
gscale =20; %goal scaler, note that 1/gscale*gdx
oscale =25; %obstacle avoidence scaler, oscale*oadx
Lscale = 50;  %LOS scale
maxvel = 5; %maximum velocity
minsep =10;   %minimum seperation between bots
goalcut = 20/gscale; %maximum distance until we say the
%robot is near enough to the goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


t =0;
cyclenum =1;
%Pre allocate matrixes to imporove computational time
numberofcycles = length([0:tstep:tfin]);
connectedness = false([robonum,robonum,numberofcycles]);
positions = zeros([2,robonum,numberofcycles]);
neargoal = numberofcycles*ones([1 robonum]);
%%%%

positions(1,:,cyclenum) = xbots;  %these come from the workspacecreator
positions(2,:,cyclenum) = ybots;

connectedness(:,:,1) = connections(positions(:,:,1),stats);
    groups = grouper(connectedness(:,:,cyclenum));
% upper = triu(connects);
% lower = tril(connects);
% chec = upper & lower';
% connectedness(:,:,cyclenum) = chec;

while(t<tfin)
    for r = 1:robonum %for each robot
        commsindx = [];
        clear xbots;
        clear ybots;
        groupnum = find(groups(:,r));  %find which group the robot belongs
        %to
        groupidx = find(groups(groupnum,:)); %find which robots are in that
        %group
        counter =0;
        for i = groupidx;
            counter = counter+1;
            xbots(counter) = positions(1,i,cyclenum);  %make xbots out of
            %that group.
            ybots(counter) = positions(2,i,cyclenum);
            if connectedness(r,i,cyclenum)
                commsindx(end+1) = counter;
            end
        end
        x = positions(1,r,cyclenum);
        y = positions(2,r,cyclenum);
```

```
P=[];
C = zeros(4,length(xbots));


%Goal velocity%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
gdx = 1/gscale*(gx(r)-x);
gdy = 1/gscale*(gy(r)-y);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (~(gdx || gdy))  %if both equal zero, the robot is at the goal
    %'robot at goal'
    P(:,1) = [.0001;.0001];  %this stops an error when the robot
    %is *exactly* at the goal which never happens
else
    P(:,1) = [gdx;gdy];  %a goal will always exist except if the
    %robots reach exactly the goal.
end

if norm([gdx;gdy]) < goalcut
    if cyclenum < neargoal(r)  %this returns a vector of when the
        %robot reaches near its goal, if it never gets near it,
        neargoal(r) = cyclenum; %the vector value is the total
        %number of cycles.
    end
end

for o = 1:length(stats) %for each obstacle

    %Obstacle avoidance%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [oadx oady] = Bishopobstacle(x, y, o, stats);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if(sqrt(oadx^2+oady^2) ~= 0)   %if there is a vector, pass
        %it to the controller
        P(:,end+1) = [oadx;oady].*oscale;
    end

    %LOS stuff%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    for q =1:length(xbots) %check LOS with each robot
        if xbots(q) ~= x %the robot will always have LOS with itself
            A = [stats(1).vx';stats(1).vy'];

            xB = xbots(q);
            yB = ybots(q);

            Ab(1,:) = A(1,:) - xB;   %translating the obstacle so B
            %is at the origin, do it can be checked by LOSchecker
            Ab(2,:) = A(2,:) - yB;

            xtrans = x -xB;
            ytrans = y -yB;

            [outside, sanctum, gammab, gammat] = ...
            LOSconnect(xtrans,ytrans,Ab);
            %is LOS needed for these robots?
            if (~sanctum) %if it is NOT in the inner sanctum
                %or outside the occlusion lines
                %then LOS needs to be evaluated.
                connxr = sum(connectedness(r,:,cyclenum)) -1;
                connxq =sum(connectedness(groupidx(q),:,cyclenum))...
                -1;  %minus one since it will always have conx with
```

```
            %itself
                if connxr < connxq %if r has less connextions than
                    %q r needs to move
                    [LOSdx,LOSdy] = ...
                        LOSvelf(xtrans,ytrans, gammab, gammat, outside);
                    %P(:,end+1) = [LOSdx; LOSdy].*Lscale;
                    %quiver(xbots(r),ybots(r),LOSdx,LOSdy, 'm')
                    if norm([LOSdx, LOSdy]) > norm([C(1,q),C(2,q)])
                        C(1,q) = LOSdx*Lscale;   %if this is a
                        %larger LOS vector use it
                        C(2,q) = LOSdy*Lscale;
                    end
                end
                %other case
                if connxr == connxq  % or if they have equal
                %connections, the lower numbered robot needs to move
                    if r < groupidx(q)
                        [LOSdx,LOSdy] =...
                            LOSvelf(xtrans,ytrans,...
                            gammab, gammat, outside);
                   %P(:,end+1) = [LOSdx; LOSdy].*Lscale;
                   %quiver(xbots(r),ybots(r),LOSdx*Lscale,LOSdy, 'm')
                        if norm([LOSdx, LOSdy]) >...
                                norm([C(1,q),C(2,q)])
                            C(1,q) = LOSdx*Lscale;   %if this is a
                            %larger LOS vector use it
                            C(2,q) = LOSdy*Lscale;
                        end
                    end
                end
            end
        end  %end if q~=r statement
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    end %end for each robot q
end  %end for each obstacle

%Range part%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for q = 1:length(xbots) %for each robot
    if x ~= xbots(q)
        dist = sqrt((xbots(q)-x)^2 + (ybots(q)-y)^2); %dist = 0 if
        %they are the same

        xunit = (xbots(q) - x)/dist;   %must be written this way
        %to ensure vector is pointing the correct way.
        yunit = (ybots(q) - y)/dist;

        if (dist < minsep)  %dist must be nonzero
            dist = dist/minsep;  %on a scale from 0 to 1 how
            %small is the distance compared to minsep
            delta =.01;
            h =.556929;
            xsig =1.1229*h;
            c = 7.05538;
            %mexhat
            zplus  = (1/(2*pi*xsig))*c*(h-(dist+delta)^2/xsig^2)...
                *exp(-1*(dist+delta)^2/(2*xsig^2));
            zminus = (1/(2*pi*xsig))*c*(h-(dist-delta)^2/xsig^2)...
                *exp(-1*(dist-delta)^2/(2*xsig^2));
            dz = (zplus-zminus)/(2*delta);
            P(:,end+1) = [xunit; yunit].*dz;  %if the distance
            %is less than minimum seperation,
        else
```

```
                dz = Improvedsemicont(dist);
                C(3,q) = dz*xunit; %velocity times unit vector
                C(4,q) = dz*yunit;
            end
        end
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %         if r ==1
    %             [M N] = size(P);
    %             for u = 1:N
    %                 quiver(xbots(r),ybots(r),P(1,u),P(2,u),'k')
    %             end
    %             [M N] = size(C);
    %             for c=1:N
    %                 quiver(xbots(r),ybots(r),C(1,c),C(2,c),'g')
    %                 quiver(xbots(r),ybots(r),C(3,c),C(4,c),'r')
    %             end
    %         end
    %         P
    %         C
    commnum = sum(connectedness(r,:,cyclenum))-1; %minus one since
    %it will have coms with itself
    %commsindx = find(connectedness(r,:,cyclenum));

    A =  paracontrollerupdate(P,C, commsindx, commnum);

    N = size(A);
    if N(2)==1;
        feasibleVel =A;
    else
        %A
        feasibleVel = smartSum(A);
    end

    dmag = norm(feasibleVel);

    if (dmag >maxvel)
        feasibleVel = maxvel*feasibleVel/dmag;
    end

    dvect(:,r) =feasibleVel.*tstep;

  end
%     xbots = xbots+dvect(1,:);
%     ybots = ybots+dvect(2,:);
%
  cyclenum = cyclenum+1;
  positions(1,:,cyclenum) = positions(1,:,cyclenum-1) + dvect(1,:);
  positions(2,:,cyclenum) = positions(2,:,cyclenum-1) + dvect(2,:);

  connectedness(:,:,cyclenum) = connections(positions(:,:,cyclenum),stats);
  groups = grouper(connectedness(:,:,cyclenum));
  t =t+tstep;
end
```

```
%simulation plotter
%removed all of the plotting elements from the simulation for comutational
%time purposes, this way I can just grab a figure and mess with it from
%there.

figure(1);
hold on;

[a b c] =size(positions);

for n = 1:b
    xpositions(1,1:c) = positions(1,n,1:c);
    ypositions(1,1:c) = positions(2,n,1:c);

    plot(xpositions,ypositions, 'b');
end
```

M Programs for baseline simulations

```
function [connectedness, neargoal] = baseline(xbots,ybots,stats,gx,gy);
%Tom Dunbar
%Baseline function, this just has the obstacle avoidance, goal and
%interrobot collision smart summed together

%run workspace creator

%Parameters%%%%%%%%%%%%%%%%%%%%%%%
robonum = 15; %the number of robots in the simulation
tfin = 500;   %the final time
tstep = .5; %the time incriment
gscale =20;  %goal scaler, note that 1/gscale*gdx
oscale =25; %obstacle avoidence scaler, oscale*oadx
Lscale = 50;  %LOS scale
maxvel = 5; %maximum velocity
minsep = 10;
goalcut = 20/gscale; %maximum distance until we say the robot is near
% enough to the goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


t =0;
cyclenum =1;
%Pre allocate matrixes to imporove computational time
numberofcycles = length([0:tstep:tfin]);
connectedness = false([robonum,robonum,numberofcycles]);
positions = zeros([2,robonum,numberofcycles]);
neargoal = numberofcycles*ones([1 robonum]);
%%%

positions(1,:,cyclenum) = xbots;  %these come from the workspacecreator
positions(2,:,cyclenum) = ybots;

connectedness(:,:,1) = connections(positions(:,:,1), stats);


while(t<tfin)
    for r = 1:robonum %for each robot

        P =[];
%Goal velocity%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        gdx = 1/gscale*(gx(r)-xbots(r));
        gdy = 1/gscale*(gy(r)-ybots(r));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (~(gdx || gdy))  %if both equal zero, the robot is at the goal
    %'robot at goal'
    P(:,1) = [.0001;.0001];  %this stops an error when the robot
% is *exactly* at the goal which never happens
        else
    P(:,1) = [gdx;gdy];  %a goal will always exist except if the
% robots reach exactly the goal.
        end

        if norm([gdx;gdy]) < goalcut
    if cyclenum < neargoal(r)  %this returns a vector of when the
% robot reaches near its goal, if it never gets near it,
neargoal(r) = cyclenum; %the vector value is the total
% number of cycles.
            end
```

```
        end

for o = 1:length(stats) %for each obstacle

    %Obstacle avoidance%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            [oadx oady] = Bishopobstacle(xbots(r), ybots(r), o, stats);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if(sqrt(oadx^2+oady^2) ~= 0)   %if there is a vector, pass it
% to the controller
                P(:,end+1) = [oadx;oady].*oscale;
            end
end  %end for each obstacle

%Range part%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for q = 1:length(xbots) %for each robot
            if r ~=q
dist = sqrt((xbots(q)-xbots(r))^2 +
 (ybots(q)-ybots(r))^2); %dist = 0 if they are the same

xunit = (xbots(q) - xbots(r))/dist;   %must be written
% this way to ensure vector is pointing the correct way.
                yunit = (ybots(q) - ybots(r))/dist;

if (dist < minsep)  %dist must be nonzero
    dist = dist/minsep;  %on a scale from 0 to 1 how small
% is the distance compared to minsep
                    delta =.01;
                    h =.556929;
                    xsig =1.1229*h;
                    c = 7.05538;
    %mexhat
    zplus  =...
(1/(2*pi*xsig))*c*(h-(dist+delta)^2/xsig^2)*exp(-1*(dist+delta)^2/(2*xsig^...
2));

    zminus =...
(1/(2*pi*xsig))*c*(h-(dist-delta)^2/xsig^2)*exp(-1*(dist-delta)^2/(2*xsig^...
2));

                    dz = (zplus-zminus)/(2*delta);
                    P(1,end+1) = dz*xunit;
    P(2,end+1) = dz*yunit;  %if the distance is less than
% minimum seperation,
                end
            end
        end


        feasibleVel =  paracontrollerbase(P);
%figure(1)
%quiver(xbots(r),ybots(r),feasibleVel(1),feasibleVel(2), 'm')

        dmag = norm(feasibleVel);

        if (dmag >maxvel)
            feasibleVel = maxvel*feasibleVel/dmag;
        end

        dvect(:,r) =feasibleVel.*tstep;

    end
```

```
    xbots = xbots+dvect(1,:);
    ybots = ybots+dvect(2,:);

    cyclenum = cyclenum+1;
    positions(1,:,cyclenum) = xbots;
    positions(2,:,cyclenum) = ybots;


    connectedness(:,:,cyclenum) = connections(positions(:,:,cyclenum),...
stats);

    t =t+tstep;
end
```

```
function feasibleVel = paracontrollerbase(P)  %P is a 2-N priority matrix
% which is in the format of rows: dx;dy
%C is a 4-N comm matrix which is in
%the format, rows: dxLOS;dyLOS;dxrange;dyrange

%Tom Dunbar
%Trident Project
%24MAR

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5
%Check if all priority 1 velocities are feasible


feasible = feasiblity(P);

A=P;
while(~feasible) %while it is not feasible
D=[];
veloA = []; %blanks it out or else an infinite loop will occur!!!
    S = size(A);
    for i = 1:S(2)
        veloA(i) = norm(A(:,i));
    end

    [velo i] = min(veloA);
    D(1,:) = [A(1,1:i-1),A(1,i+1:end)];  %delete the smallest velocity
    D(2,:) = [A(2,1:i-1),A(2,i+1:end)];

    feasible = feasiblity(D); %recheck feasibility
    A =D; %after the smallest has been deleted update A;

    %note that velocities are sucessively deleted, and thus feasibility
    %will eventually be acheived because soon there will only be two
    %velocities left.
end
%A
S = size(A);
if S(2)==1;
    feasibleVel =A;
    return
```

```
end

%Now that we have a set of feasible vectors its time to create the final
%motion vector!!!!
feasibleVel = smartSum(A);
```

N Programs for quantitative analysis

```
%Tom Dunbar
%Trident Project
%27MAR
%Note: desired simulation must be changed manually and desired number of
%robots must also be changed manually in paracontrollerupdate.m

clc
clear

Agroupnum = zeros([30 1001]); %pre-allocate memory for all data
Alinktot = zeros([30 1001]);
Aetot = zeros([30 1001]);
Adegreemin = zeros([30 1001]);
Asimp = zeros([1 30]);
Aneargoal = zeros([30 15]);

Aavggroupnum = zeros([1 1001]); %pre-allocate memory for average stuff
% that will be plotted
Aavglinktot = zeros([1 1001]);
Aavgetot = zeros([1 1001]);
Adegreemin = zeros([1 1001]);


for w = 1:30
    w
    %load a workspace
    wspace = ['workspace', num2str(w)];
    load(wspace);
    %run controller
    [connectedness, neargoal] = fullsim(xbots,ybots,stats,gx,gy);
    %run analysis
    [groupnum, linktot, etot, degreemin, simp] = kconnect(connectedness);
    [a b c] = size(connectedness);
    %save data for a controller
    Agroupnum(w,:) = groupnum;
    Alinktot(w,:)  = linktot;
    Aetot(w,:)     = etot;
    Adegreemin(w,:)= degreemin;
    Asimp(w)  = sum(simp)/c; %percent of time in simple redundancy
    Aneargoal(w,:) = neargoal;
end

%plotting matrixes
Aavggroupnum = sum(Agroupnum)./30;
Aavglinktot = sum(Alinktot)./30;
Aavgetot = sum(Aetot)./30;
Aavgdegreemin = sum(Adegreemin)./30;


%the 6 measures of effectiveness
Aavgneargoal = sum(sum(Aneargoal')./15)/30  %avgerage time each robot took
% to complete goal, then average time through the swarm
Aavgsimp = sum(Asimp)/30  %percentage of time in simple redundancy
AAreagroupnum = sum(Aavggroupnum);
AArealinktot = sum(Aavglinktot);
AAreaetot = sum(Aavgetot);
AAreadegreemin = sum(Aavgdegreemin);

%plot everything
figure(4)
```

```
hold on;
subplot(2,2,1)
plot(Aavggroupnum)
axis([0 1000 0 5])
Ts = ['Number of groups (avg. over 30 runs)- Area = ',...
num2str(AAreagroupnum)];

xlabel('Time')
ylabel('# of Groups')
title(Ts)

subplot(2,2,2)
plot(Aavglinktot)
axis([0 1000 0 15])
Ts = ['Number of links/robot (avg. over 30 runs) - Area =',...
num2str(AArealinktot)];

xlabel('Time')
ylabel('links/robot')
title(Ts)

subplot(2,2,3)
plot(Aavgetot)
axis([0 1000 0 7])
Ts = ['Algebraic connectivity (avg. over 30 runs) - Area =',...
num2str(AAreaetot)];

xlabel('Time')
ylabel('e_2')
title(Ts)

subplot(2,2,4)
plot(Aavgdegreemin)
axis([0 1000 0 7])
Ts = ['Minimum degree (avg. over 30 runs) - Area =',...
num2str(AAreadegreemin)];

xlabel('Time')
ylabel('Links')
title(Ts)


save goal1
```

```
function [groupnum, linktot, etot, degreemin, simp] =...
kconnect(connectedness)
...

%Tom Dunbar
%23 FEB
%Trident Project

%This computes the statistics of a graph namely K-connectedness and vertex
%degree
```

```
[a b c] = size(connectedness);
for cycle = 1:c;
adja = connectedness(:,:,cycle);
Vdegree = sum(adja);  %vertex degree

links = Vdegree-1;   %number of links per robot

for i = 1:length(Vdegree)
    h(i,i) = Vdegree(i);
end

L = -1*adja +h;  %the laplacian matrix
e = eig(L); %eigienvalues of the laplacian
e = sort(e);  %put them in increasing numerical order
groups = size(grouper(adja));

groupnum(cycle) = groups(1);   %number of groups
linktot(cycle) = sum(links)/length(links); %average number of links per
% robot
etot(cycle) = e(2);  %algebraic connectivity
degreemin(cycle) = min(links); %minimum vertex degree
if min(links) >=2
    simp(cycle) = 1; %simple redudancy check
else
    simp(cycle) = 0;
end
% if min(links) >= desiredlinks
%     meating = 1;  %is the controller meeting the minimum number of links
% per robot?
% else     %only applies to desiredlinks other than 2
%     meating =0;   %simp will return the same info if desiredlinks =2
% end
end
% figure(3)
% clf
% hold on;
% plot(groupnum, 'k-')
% plot(etot,'-')
% plot(ktot,'r-')
% plot(linktot, 'g-')
% axis([0 100 0 15])
```

```
function groups = grouper(adjacency)
%Tom Dunbar
%Trident Project
%17FEB

%this function takes an adjanecy matrix and using a depth first search
%returns the groups the robots are in.

%start at the first robot (first row)

counter =0;
```

```
zeroidx =1; %just to get things started

while(length(zeroidx))
    unexp = zeroidx(1); %takes the first new connection as unexplored
    group = zeros([1,length(adjacency)]); %initialize group
    while(length(unexp))  %while there still remains unexplored
% connections
        r = unexp(1);

        connex = find(adjacency(r,:));
        group(r) =2;

for i = connex %for each connection
    if (~group(i))  %if group(i) was previously unconnected
                group(i) = 1;
            end
        end

        unexp = find(group ==1);
    end  %end while loop

    counter = counter +1;
    groups(counter, :) = group; %groups should only be 2s and zeros

    accountability = sum(groups,1);  % sums all the columns; in the end
% should be a full matrix of twos

    zeroidx = find(~accountability); %finds the zeros in the
% accountability
end
```

```
function connects = connections(positions,stats)
%Tom Dunbar
%21FEB

%Connectivity Checker for the restricted controller, this takes the
%positions matrix which is a record of all the postions and determines
%which bots are connected. This information builds the connectedness
% matrix

minsep = 10;
maxsep = 50;
robonum = 15;

connects = logical(eye(robonum));

for r = 1:robonum
    for i = 1:robonum
        if i ~= r
    dist = sqrt((positions(1,i) - positions(1,r))^2...
+(positions(2,i) - positions(2,r))^2);


    if (dist < maxsep) %first check if it is in range
```

```
%Next check LOS%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                oldvalue =true;
for o =1:length(stats) %check each obstacle
                    A = [stats(o).vx';stats(o).vy'];
                    clear Ab
                    xB = positions(1,i);
                    yB = positions(2,i);

    Ab(1,:) = A(1,:) - xB;   %translating the obstacle so
% B is at the origin,
                    Ab(2,:) = A(2,:) - yB;

                    xtrans = positions(1,r) -xB;
                    ytrans = positions(2,r) -yB;
    [outside, sanctum, gammab, gammat] =...
LOSconnect(xtrans,ytrans,Ab);

    oldvalue = sanctum & oldvalue; %if any of these is a
% 0, they will forever be a zero
                end
                if (oldvalue)
                    connects(r,i) = true;
                end

    end %ends range check
end%ends if r ~=i
    end
end
```

```
%Tom Dunbar
%Trident project, data analisis
%13APR
load baselin;
desired =0;


%avg transients
SSlink(1) = sum(avglinktot(1:400))/400;
SSe2(1)  = sum(avgetot(1:400))/400;
SSmindegree(1) = sum(avgdegreemin(1:400))/400;

allgoal(1) = avgneargoal;

for desired = 1:8
    workspae = ['goal',num2str(desired)];
    load(workspae);

    %avg transients
    SSlink(desired+1) = sum(Aavglinktot(1:400))/400;
    SSe2(desired+1)  = sum(Aavgetot(1:400))/400;
    SSmindegree(desired+1) = sum(Aavgdegreemin(1:400))/400;

    allgoal(desired+1) = Aavgneargoal;
```

```
        workspae = ['comms',num2str(desired)];
    load(workspae);

    %avg transients
    Clink(desired+1) = sum(Aavglinktot(1:400))/400;
    Ce2(desired+1)  = sum(Aavgetot(1:400))/400;
    Cmindegree(desired+1) = sum(Aavgdegreemin(1:400))/400;

    Callgoal(desired+1) = Aavgneargoal;
end

gainlink = (SSlink - SSlink(1)) ./SSlink(1)*100;
gaine2 = (SSe2 - SSe2(1)) ./SSe2(1)*100;
gainmindegree = (SSmindegree - SSmindegree(1))...
    ./SSmindegree(1)*100;
gaingoal = (allgoal - allgoal(1)) ./allgoal(1)*100;

Cgainlink = (Clink - SSlink(1)) ./SSlink(1)*100;
Cgaine2 = (Ce2 - SSe2(1)) ./SSe2(1)*100;
Cgainmindegree = (Cmindegree - SSmindegree(1))...
    ./SSmindegree(1)*100;
Cgaingoal = (Callgoal - allgoal(1)) ./allgoal(1)*100;

Cgaingmindegree(1) = gainmindegree(1) ;
Cgainge2(1) = gaine2(1) ;
Cgaingoal(1) = gaingoal(1) ;

Cmindegree(1) = SSmindegree(1) ;
Ce2(1) = SSe2(1) ;
Callgoal(1) = allgoal(1) ;


figure(1)
clf
subplot(3,1,1)
hold on;
plot([0:8], gaine2, 'k *')
plot([0:8], Cgaine2, 'k ^')
grid on
title('Average percentage gain in e_2 over baseline')
xlabel('desired number of links/robot')
ylabel('Percentage')
axis([0 8 -50 350])

subplot(3,1,2)
hold on;
plot([0:8], gainmindegree, 'k *')
plot([0:8], Cgainmindegree, 'k ^')
grid on
title('Average percentage gain in minimum degree over baseline')
xlabel('desired number of links/robot')
ylabel('Percentage')
axis([0 8 -50 350])

subplot(3,1,3)
hold on;
grid on;
plot([0:8], gaingoal,'k *');
plot([0:8], Cgaingoal,'k ^');
grid on;
title('Average percentage gain in time to goal over baseline')
xlabel('desired number of links/robot')
```

```
ylabel('Percentage')
axis([0 8 -10 60])
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure(2)
clf
[AX,H1,H2] = plotyy([0:8],SSmindegree, [0:8], allgoal);


set(H1,'Marker','*');
set(H1,'LineStyle','none');
get(AX(1)); %axis one for k connex
hold on;
plot([0:8], SSe2,'*');
plot([0:8], Ce2, '^');
plot([0:8], Cmindegree, '^');
axis([0 8.5 0 5]);
%
%
get(AX(2));
%hold on;
plot([0:8], Callgoal, '.-');
%axis([0 8.5 0 500])
%
%
% % %Ts = ['Number of links/robot (avg. over 30 runs) - Area =',
% num2str(AArealinktot)];
% %xlabel('Desired number of links')
% % ylabel('links/robot')
% % %title(Ts)
% %
%
%set(get(AX(1),'Ylabel'),'String','K-Connectivity')
%set(get(AX(2),'Ylabel'),'String','Time to Goal')
```

```
function createfigure(x1, y1, y2, y3, y4, y5, y6)
%CREATEFIGURE(X1,Y1,Y2,Y3,Y4,Y5,Y6)
%  X1: vector of x data
%  Y1: vector of y data
%  Y2: vector of y data
%  Y3: vector of y data
%  Y4: vector of y data
%  Y5: vector of y data
%  Y6: vector of y data

%  Auto-generated by MATLAB on 13-Apr-2005 21:59:24

%% Create figure
figure1 = figure('FileName','C:\Documents and...
Settings\Administrator\Desktop\Dunbarstuff\compareboth.png','PaperPosition...
',[-1.083 2.031 10.67 6.938]);
```

```
%% Create axes
axes1 = axes(...
  'YColor',[0 0 1],...
  'YTick',[0 1 2 3 4 5 5],...
  'Parent',figure1);
axis(axes1,[0 8.5 0 5]);
xlabel(axes1,'Desired number of links/robot');
ylabel(axes1,'K-Connectivity');
grid(axes1,'on');
hold(axes1,'all');

%% Create plot
plot1 = plot(...
  x1,y1,...
  'LineStyle','none',...
  'Marker','*',...
  'Parent',axes1);

%% Create plot
plot2 = plot(...
  x1,y2,...
  'LineStyle','none',...
  'Marker','*',...
  'Parent',axes1);

%% Create plot
plot3 = plot(...
  x1,y3,...
  'LineStyle','none',...
  'Marker','^',...
  'Parent',axes1);

%% Create plot
plot4 = plot(...
  x1,y4,...
  'LineStyle','none',...
  'Marker','^',...
  'Parent',axes1);

%% Create plot
plot5 = plot(...
  x1,y5,...
  'Marker','.',...
  'Parent',axes1);

%% Create plot
plot6 = plot(...
  x1,y5,...
  'Marker','.',...
  'Parent',axes1);

%% Create axes
axes2 = axes(...
  'YColor',[0 0.5 0],...
  'YAxisLocation','right',...
  'YTick',[200 250 300 350 400 450 450],...
  'Parent',figure1);
axis(axes2,[0 8.5 200 450]);
title(axes2,{'Comparison of original and comm. varient controller ','with...
respect to goal completion and robustness of network'});

ylabel(axes2,'Time ot goal (avg.) in interations of controller');
```

```
hold(axes2,'all');

%% Create plot
plot7 = plot(...
  x1,y6,...
  'Parent',axes2,...
  'DisplayName','Original');

%% Create plot
plot8 = plot(...
  x1,y5,...
  'LineStyle','--',...
  'Parent',axes2,...
  'DisplayName','Comm varient');

%% Create legend
legend1 = legend(...
  axes2,{'Original','Comm varient'},...
  'Color',[0.8 0.8 0.8],...
  'Position',[0.1584 0.8152 0.1338 0.06306]);

%% Create textbox
annotation1 = annotation(...
  figure1,'textbox',...
  'Position',[0.3115 0.7132 0.1592 0.2042],...
  'FitHeightToText','off',...
  'String',{'Triangles mark upper and','lower bounds for','K-...
connectivity','of the Comm variant.','While * mark upper and','lower...
bounds of the','Original controller'});
```

(THIS PAGE INTENTIONALLY LEFT BLANK)